VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2020 Vojtěch Prokop

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Vojtěch Prokop**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: ATLAS consulting spol. s r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Marek Běhálek, Ph.D.**

Konzultant bakalářské práce: Mgr. Tomáš Řehák

Datum zadání: 01.09.2019

Datum odevzdání: 30.04.2020

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky 1D16 OSTRAVA

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravč 21. dubna 2020

Prokop Vojtěch

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 27. dubna 2020

ATLAS consulting spol. s r.o.
Výstavní 292/13
702 00 Ostrava, Moravská Ostrava
IČO 46578706, DIČ CZ46578706

Rád bych na tomto místě poděkoval všem, kteří mi s touto prací pomohli. Především společnosti Atlas consulting spol. s r.o. za možnost absolvování praxe, a také panu Ing. Marku Běhálkovi, Ph.D., jakožto vedoucímu, za strávený čas u konzultací.

Abstrakt

Tato práce popisuje absolvování individuální odborné praxe ve společnosti Atlas consulting s r.o, kde jsem pracoval jako člen frontendového týmu. Práce se zabývá od charakteristiky vyvíjeného softwaru Codexis, až po rozebrání řešených úkolů a s nimi spojenými technologiemi. Závěrem v této práci nalezneme hodnocení celkového působení na praxi, kde jsou shrnuty znalosti jak získané před praxí na akademické půdě, tak i znalosti nabyté na praxi.

Klíčová slova: JavaScript, TypeScript, React, webová aplikace

Abstract

This thesis describes the completion of individual professional practice in the company Atlas consulting s r.o., where I worked as a member of the frontend team. The thesis deals from the characteristics of the developed Codexis software, to the analysis of the solved tasks and related technologies. In conclusion, in this thesis we find an evaluation of the overall impact on practice, which summarizes the knowledge both acquired before the practice on university and the knowledge acquired in practice.

Keywords: JavaScript, TypeScript, React, web application

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
BE	–	Backend
CLI	–	Command Line Interface
CSS	–	Cascading Style Sheets
E2E	–	End-to-End
FE	–	Frontend
HoC	–	Higher-Order Component
JS	–	JavaScript
P2P	–	Peer-to-peer
TS	–	TypeScript
UI	–	User Interface
UX	–	User Experience
WebRTC	–	Web Real-Time Communication

Seznam obrázků

Seznam tabulek

1 Úvod

Tato práce se bude zabývat popisem mé odborné praxe. Krátkým seznámením s firmou, využitými technologiemi a charakteristikou řešených úkolů.

Individuální odbornou praxi jsem si zvolil za účelem poznání, jaké to je, být začleněn do týmu, který pracuje na opravdovém projektu. Snahou bylo připojit se k řešení každodenních problémů u vývoje komplexních systémů a tím nabýt praktické zkušenosti, které mi doposud scházely. Výběr firmy jsem prováděl s úmyslem participace na vytváření produktů nejmodernějšími technologiemi.

Po úspěšně zvládnutém pohovoru jsem nastoupil na pozici vývojáře webových aplikací ve firmě Atlas consulting spol. s r.o. Zde jsem se podílel na úpravách a rozvoji systému Codexis, rozvoji knihovny React-Atlantic a dalších menších projektech. Prezentační vrstva těchto projektů byla vyvíjena v jazyce JavaScript za použití veřejně publikované knihovny React.

Úvodem bych ještě rád krátce nastínil obsah jednotlivých kapitol. V kapitole druhé představím samotnou firmu Atlas consulting a její produkty. Kapitola třetí vylíčí rozsah využitých technologií. Dále v kapitole čtvrté se budu zabývat mou rolí ve firmě a pracovní náplní, jakožto člena frontendového týmu. V následující kapitole, páté, jsou popsány úkoly, na kterých jsem se za dobu své praxe podílel.

Před závěrem práce dojde k shrnutí celé praxe. Především budou zhodnoceny teoretické a praktické znalosti získané v průběhu studia, ale i znalosti, které mi scházely. Celou práci uzavře krátký závěr, ve kterém popíši množinu všech dovedností a zkušeností nabytých během praxe.

2 Seznámení s firmou ATLAS consulting spol. s r.o.

Atlas consulting spol. s r.o. [1] je jeden z členů skupiny Atlas Group, mimo jiné zde patří Atlas software a odborné nakladatelství vydávající právnickou literaturu Codexis publishing. Atlas consulting je ryze českou softwarovou společností, přinášející na trh právní a manažerské informační systémy a aplikace. Na trhu působí, jíž od roku 1992.

Oproti ostatním členům Atlas Group, Atlas consulting sídlí pouze v Ostravě. Pro zbylé členy platí, že je můžeme najít jak v Praze, tak i Brně. Firma je středně velká, strukturovaná do několika týmů. Momentálně se společně věnuje vývoji systému Codexis a Econix.

2.1 Codexis

Codexis je bezesporu [2] nejvýznamnějším produktem, který patří mezi nejrozšířenější právní informační systémy v České republice. Tento systém čerpá veškerý svůj obsah z věrohodných zdrojů, např. Sbírky zákonů, Sbírky mezinárodních smluv a tím obsahuje zákony spadající do legislativy České republiky, legislativy Evropské unie, judikatury České republiky a judikatury Evropské unie.

Mezi uživatele přívětivého informačního systému Codexis patří většinou lidé pracující v právnickém sektoru: ekonomové, účetní, úředníci, právníci, manažeři. Efektivně jsou těmto uživatelům prezentovány pohledy na legislativu, jež se podrobuje stovkám změn za rok, a proto je těžké se v ní vyznat. Software nabízí co nejjednodušší vyhledávání, podle údajů o těchto právních dokumentech, dle typu dokumentu, autora, zdroje, účinnosti, či klíčových slov. Veškeré dokumenty jsou navíc uvedeny v aktuálním, minulém i budoucím znění.

Software je možné sestavit na míru potřebám konkrétního uživatele, prostřednictvím řady doplňků a nástrojů např. právní kalkulačky, umožňující snadné a přehledné výpočty nebo komentáře liberis či právní slovníky obsahující právní pojmy s judikaturou a právnický latinský slovník.

Codexis sází na inovativní přístup. Neustále je zdokonalován, vyvíjen v souladu aktuálních trendů a přizpůsobován uživatelům za účelem uživatelského komfortu. Je navržen tak, aby práce s ním byla intuitivní, přehledná a snadná na orientaci.

2.2 Econix

Druhým produktem je manažerský informační systém Econix. Do tohoto systému patří Equanta, software, jenž poskytuje obraz o financích, analýzu, a různé výpočetní funkce. Smlouvy, aplikace pro evidenci a správu smluv a dokonalý přehled ve všech smlouvách. Hlídač ochranných známek, aplikace, která pomocí nejmodernějších algoritmů hlídá známky, porovnává, sleduje a v případě shody vás upozorní na kolizi. A další software produkty jako Manažer datových schránek, Navigal, Evidence smluv a Kontrola insolvence.

3 Použité technologie

3.1 TypeScript

Programovací jazyk TypeScript [3], nadmnožina JavaScriptu, je kompilován do čistého JavaScriptu. Jak už z názvu vypovídá, obohacuje JavaScript o typované proměnné, čímž získáme z dynamicky typovaného jazyku striktně typovaný jazyk (Java, C++, C). V základu je úroveň typovosti nastavena na volitelnou, ačkoliv přes konfigurační soubory můžeme nastavit striktnější chování, poskytující lepší kontrolu nad zdrojovým kódem, dokumentaci kódu a hlavně validaci kódu v reálném čase při vývoji.

3.2 React

React [4] je knihovna specializující se na tvorbu uživatelských rozhraní. Při použití Reactu je uživatelské rozhraní členěno do flexibilních, znovupoužitelných komponent, zdrojový kód je psán deklarativním způsobem a je využita JSX syntaxe.

Vývojář je při tvorbě uživatelského rozhraní veden k popisu **jak by měla stránka vypadat**, aniž by musel řešit **jak bude stránka překreslena**. Mimo jiné toto je také rozdíl mezi **deklarativním** a **imperativním** programováním.

3.2.1 Komponenty

Komponenty si lze představit jako malé části, které jsou základem pro vytvoření větších modelů. Každá komponenta může obsahovat další komponentu, a tak vytvářet stromovou strukturu. Je důležité vytvářet co nejmenší komponenty, řešící malou část většího celku. Propojením těchto malých částí vytvoříme komplexnější uživatelské rozhraní.

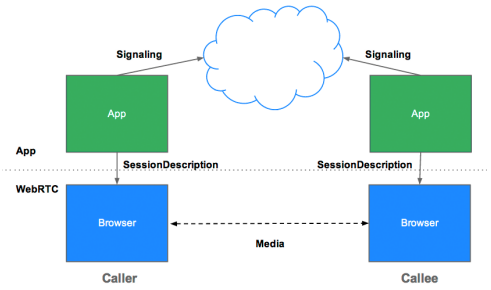
Vstupními parametry komponenty jsou vlastnosti, které nalezneme v zdrojovém kódu ve zkráceném tvaru **props**, objekt z rodičovské komponenty, reprezentující data. Vlastnosti jsou pro komponentu neměnná data, která by neměla být jakýmkoliv způsobem modifikována. Pro programátora se jeví jako data, která můžeme přečíst, ale ne nastavit, neboli mutovat. Mutace vlastností povede k přímé úpravě virtuálního objektového modelu dokumentu, což může záporně ovlivnit výkon Reactu ve vykreslovacím řetězci. Rodičem způsobená změna vlastností má za následek zavolání vykreslovací funkce v komponentě.

Kromě vyskytujících se neměnných dat v podobě objektu **props**, má komponenta svůj stav. Interní stav, spravován komponentou, je různě modifikovatelný. Každou změnou stavu, jak v případě vlastností, je také zavolána vykreslovací funkce.

Zavoláním vykreslovací funkce se vrátí pro **vstup** a **stav** komponenty nový React Element, popis toho, co se má vykreslit, neboli uživatelské rozhraní reprezentující výstup komponenty.

3.3 WebRTC

Anglicky *Web Real-Time Communication* [5], je aplikační rozhraní, které vytváří mezi dvěma klienty **P2P** spojení. WebRTC podporuje přenos zvuku, videa i surových dat. Obvyklým příkladem je vytvoření služby pro video komunikaci mezi dvěma, ale i více účastníky. Dalším příkladem jsou streamovací služby. Vše je možné bez použití externích pluginů.



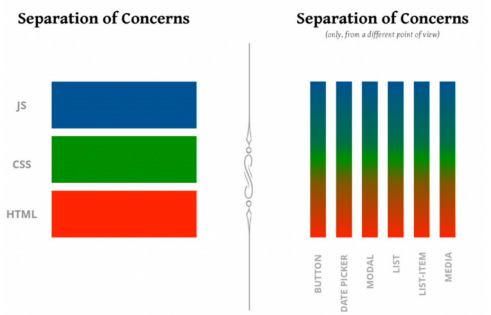
Obrázek 1: Signalizační diagram [6]

3.4 Socket.IO

Socket.IO je knihovna [7], která poskytuje komunikaci přes síťový soket. Komunikace klientů je založena na *event-based* principu. Výhodou bývá možnost persistentního spojení, což vylučuje nutnost opakovaného navazování spojení. Skládá se ze dvou částí. Obvyklým scénářem bývá na straně serveru Node.js, který vyčkává na připojení JS klienta (tím může být znova Node.js klient).

3.5 Styled Components

Využití techniky CSS v JS, stylované komponenty [8], které umožňují psaní kaskádových stylů do zapouzdřených komponent. Ve spojení s knihovnou React, je pak zásadní výhodou vývoj komponent jako modulů nezávislých na ostatních kaskádových stylech v aplikaci. V porovnání s klasickými kaskádovými styly je při vývoji aplikován jiný úhel pohledu (viz obrázek 2).



Obrázek 2: Stylované komponenty [9]

3.6 GraphQL

GraphQL [10] je dotazovací jazyk pro moderní aplikace. Klientem specifikované textové řetězce s dotazem jsou interpretované serverem, což vrací data v definovaném formátu. GraphQL přináší kompletní a srozumitelný popis dat, díky čehož klient získává možnost vyžádat přesně ta data, která chce. To umožnuje jednodušší vývoj API ve spojení s posunutím kvality aplikací na novou úroveň.

3.7 Apollo Client

Apollo Client [11] je knihovna poskytující kompletní správu stavu v JS aplikacích. Pomocí něj stačí specifikovat GraphGL dotaz, a Apollo Client zařídí dotázání backend služeb, což vrátí data, data jsou uložena do mezipaměti a zároveň všechny UI komponenty aktualizovány.

3.8 Codegen

Codegen [12], CLI nástroj, který pomáhá automaticky vygenerovat fragmenty zdrojového kódu. Odstraňuje nutnost explicitního psaní typů a metod spojených s backend službami. Ty jsou implicitně vygenerovány po zanalyzování a následném zpracování předem definovaného GraphQL schématu.

3.9 React Spring

Nástroj React Spring [13] přináší fyzikou poháněné animace, na které v dnešních uživatelských rozhraních častokrát narazíme. Tato pomůcka poskytuje dostatečnou flexibilitu a možnost tvorby, ničím neomezených, pohyblivých uživatelských rozhraní, které drasticky neovlivní výkon vyvíjené aplikace.

3.10 React Use Gesture

React Use Gesture [14] umožňuje implementaci různých pokročilých interakcí s uživatelským rozhraním, a to za pomocí pár řádků zdrojového kódu. V knihovně je možné najít metody jako `useDrag`, `useScroll`, `useWheel`, díky kterých lze jednoduše implementovat široké spektrum případu užití, a to i ve spojení s animovanou knihovnou jako React Spring nebo React Three Fiber.

4 Pracovní náplň

Po úspěšném pohovoru jsem byl přijat na pozici frontendového vývojáře ve firmě Atlas consulting s r.o. Většina zadaných úkolů byla spojena se zásahem do prezentační vrstvy jednoho z hlavních produktů, aplikace Codexis. Tyto úkoly cílily na rozšíření funkčnosti stávajícího systému, především ale úpravou jejich funkčních celků, které byly potřeba předělat. Mnohdy přidání nové funkcionality mělo za následek poškození staršího funkčního celku. Situace byla natolik kritická, že bez onoho předělání nebylo možné projekt dále rozvíjet. Předělání softwaru obnášelo kromě nového návrhu, předělání starého neresponzivního designu, do nového, plně kontrolovatelného responzivního designu.

Použitím knihovny React členíme uživatelské rozhraní do znovupoužitelných komponent. Toto vedlo frontendový tým k vytvoření knihovny komponent React-Atlantic. Komponenty umístěné v knihovně byly vytvářeny s ohledem na maximální využití skrz více projektů ve firmě. To vede k vyšší abstraktnosti komponent. Knihovna je samostatný modul, na kterém jsou projekty závislé. Tím jsme zajistili sdílení stylů i části logiky. Úkoly spojené s předěláním softwaru zahrnovaly často práci na knihovně komponent. Kompomentu jsem vždy vytvářel s důrazností na vývojářský komfort.

4.1 Organizace vývoje

V průběhu praxe jsem byl ve styku s agilním vývojem projektu. Tento inkrementálně iterativní způsob popisuji níže, zároveň s popisem mé pracovní pozice jakožto člena týmu.

4.1.1 Plánování a design

Úkoly mi byly přidělovány v závislosti na prioritě problému a ostatních členech týmu. Velikost frontendového týmu tvořila skupina 4 členů (včetně mě), 3 vývojáři s orientací na React a interní grafik, který se staral mimo grafiku i o kaskádové stylování. Všichni členové skupiny se podíleli společně se mnou na předělávání systému. Vedoucí týmu, mně a ostatním členům přiřadil část uživatelského rozhrání, kterou jsme se měli zabývat.

Práci na úkolu jsem začínal detailním rozborem s vedoucím, který mi představil samotný problém a mimo jiné vyžadoval můj časový odhad k řešení úkolu. Následně jsem si musel vyhranit dostatečný časový interval na promyšlení. Před samotnou implementací jsem návrh konzultoval. Zpočátku praxe byly mé návrhy vedoucím upravovány, tak abychom se vyhnuli špatnému návrhu, což byl hlavní kritický bod každé části aplikace před předěláním (refactoringem). První z úkolů byly z mé strany zle navrhnuty, a proto byly zásahy od vedoucího větší, to přisuzuji neznalosti technologie. Také přichází v potaz velká komplexnost aplikace, která byla pro mě matoucí. Časem byly špatné návrhy redukovány, a proto jsem měl možnost začít implementaci bez konzultace s vedoucím.

Každý z těchto úkolů bych rozdělil na dva druhy problémů. Logickou část, která je spojena s transformací dat, výpočty nad daty a uložení dat. Tato část obnášela konzultaci s vedoucím. Druhou části je uživatelské rozhraní, z pohledu kaskádových stylů, CSS. Stylistická stránka věci byla většinou konzultována s interním grafikem. Ten předkládal vytvořené návrhy s responzivním designem pro stolní počítač, laptop, tablet a mobilní zařízení.

4.1.2 Implementace

Během implementace daného úkolu jsem byl veden ke správě kódu ve verzovacím systému Git. Ten zjednodušuje spolupráci ve vývoji zejména ve vícečlenném týmu. U vývoje složitého projektu jako je Codexis se skvělé hodí. Paralelně jsem spolupracoval s týmem ve své větvi, podvětvi aktuálně řešeného segmentu systému, která byla označena číslem a krátkým popisem mého zadání. Repositář jsem udržoval po celý čas synchronizovaný s ostatními členy, abych se vyhýbal zbytečným konfliktům.

Ukončenou implementací daného úkolu, následovalo posouzení kódu s vedoucím týmu za účelem najít programátorské chyby. Posouzením kódu se předcházelo chybám, které autor kódu přehlédl při vývoji, čímž se zvýšila kvalita softwaru.

4.1.3 Testování

V poslední fázi vývoje za pomocí programovacího jazyka Java probíhalo testování. End-to-End (E2E) a testy, v kterých docházelo k vyfocení aplikace, podlehlo celé nové uživatelské rozhraní ve všech responzivních rozlišeních. Tímto se limitovala chybovost ve finální verzi. K vytvoření těchto automatických testů jsem využíval knihovnu Selenium a Cucumber. Musel jsem tedy popsat testovací scénáře mnoha případu užití.

4.1.4 Nasazení

Otestovaná verze předělaného oddílu aplikace byla vydána do produkce. Oddíl byl nadále monitorován, aby v případě výskytu nového problému byl v co nejkratším časovém úseku opraven.

5 Řešení zadaných úkolů

V této kapitole rozeberu 5 nejpodstatnějších úkolů, na kterých jsem v průběhu praxe pracoval. Především se jedná o úkoly, které obnášely podstatnější zásah do zdrojového kódu aplikace.

Každá z podkapitol obsahuje popis specifikace požadavků vybraného úkolu, analýzu a návrh a končí krátkým implementačním popisem. K jednotlivým úkolům se zároveň vztahuje časová náročnost zobrazena v tabulce č. 1.

Tabulka 1: Časová náročnost

Úkol	Počet pracovních dní
Video chat	**20**
Úprava záložky - Titulní stránka	**10**
Knihovna komponent	-
Úprava záložky - Vyhledávání	**12**
Widget prezentující novinky a aktuality	**11**

Časová náročnost strávená u vývoje knihovny React-Atlantic nelze zcela jasně odhadnout. Strávený čas byl závislý na potřebě vytvoření nové obecné komponenty, ty byly vyráběny v čase, kdy byla určitá záložka navrhována z pohledu struktury komponent.

Dále bych rád zmínil i některé nepopsané úkoly jako je přepis záložky Dokument, práce na online advokátní platformě, mnoho úprav spojených s zjednodušením zdrojového kódu a menších modifikací uživatelského rozhraní, oprav jak rozmanitých, tak i menších chyb, a nakonec integraci animací do některých vybraných částí, z důvodu lepšího uživatelského požitku.

5.1 Video chat

V rámci rozšiřování poskytovaných funkcionalit chce společnost Atlas Group nabídnout možnost online konzultace. Přidáním možnosti video komunikace přímo do systému povede k zefektivnění spolupráce mezi zákazníky hledající odbornou pomoc v právních sektorech a odborníky, kteří tuto pomoc a poradenství poskytují.

Narozdíl od Skypu, vytvořením aplikace založené na veřejně dostupné technologii WebRTC není nutné uživatelem instalace softwaru třetích stran. Uživatel bude moci využít funkčnosti online konzultace skrz webový prohlížeč, což mimo jiné odstraňuje nutnost instalace nových aktualizací. Požadavky potencionálních klientů mohou v budoucnu přibývat, a proto vyhnutí softwaru třetích stran umožňuje na míru lépe přizpůsobovat vyvíjený software požadavkům klientů.

Z nové funkcionality pak budou benefitovat obě strany, jak zákazníci, kteří jednoduchými kroky budou moci specifikovat dotaz a následně se spojit s odborníkem. Tak i odborníci, kteří budou poskytovat své služby, v systému uvidí všechny dotazy od zákazníků, na které budou

moci reagovat a domluvit si online konzultaci. U konzultace budou moci využít všechny pohledy na legislativu dostupné skrz systém Codexis, kde vše bude moci být zákazníkovi sdíleno.

5.1.1 Specifikace požadavků

Cílem úlohy bylo vytvořit webovou aplikaci, prostřednictvím které spolu budou moci dva klienti komunikovat v reálném čase s možností použití kamery, mikrofonu a také klasického posílání textových řetězců.

S úmyslem využití finální aplikace, jako výchozí stav, v nově plánovaném produktu firmy bylo požadavkem úkol vypracovat společně s dokumentací, včetně osobních poznatků. Později při vývoji byla množina požadavků rozšířena o funkčnost sdílení obrazovky a možnost uložení zvoleného datového toku dat.

5.1.2 Analýza a návrh

Jak lze vidět na obrázku č. 1., předtím než spolu klienti začnou komunikovat, je nezbytné, aby si vyměnili signalizační informace a tím navázali P2P spojení, což znamenalo kromě vytvoření uživatelského rozhraní i vytvoření signalizačního serveru. V mém případě bylo nutné navrhnout rozhraní akcí, které signalizační server bude poskytovat. Zjednodušený proces komunikace s akcemi znázorňuje UML sekvenční diagram na obrázku č. 3.

Na straně klienta byla implementace rozdělena na dvě části, a to logickou a vizuální, popisující uživatelské rozhraní.

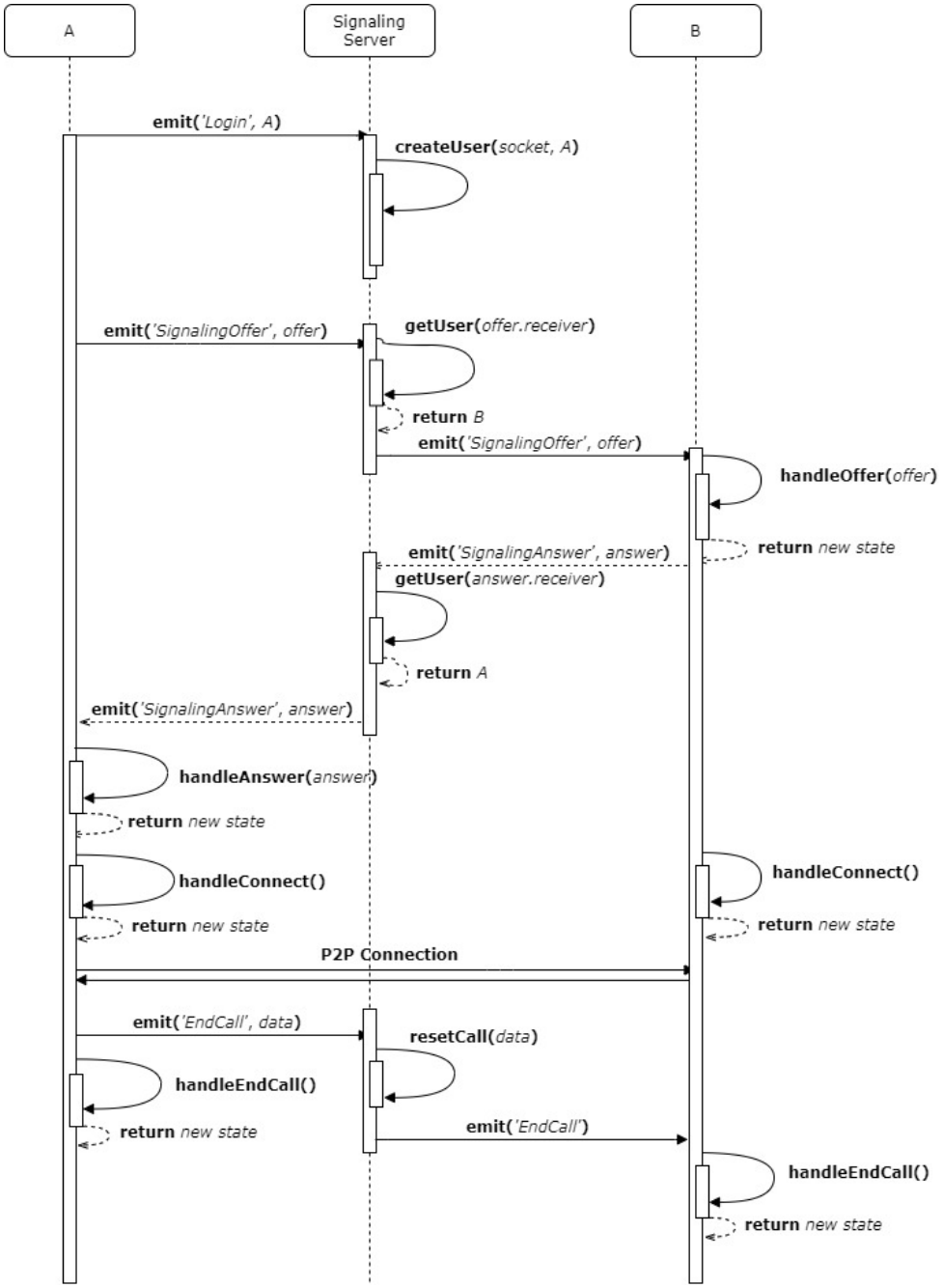
Logická část aplikace měla být vytvořena nezávisle na uživatelském rozhraní. Struktura logické části byla rozdělena (viz obrázek č. 4), do několika funkcionálních komponent. Každá komponenta poskytovala rozhraní využitím API `useContext`, což umožnovalo sdílení dat bez nutnosti explicitního vnoření vlastnosti. Závěrečné využití mělo být ve formě obalení části zdrojového kódu s předáním doménové adresy signalizačního serveru.

Z požadavků plynulo, že vizuální část, neboli uživatelské rozhraní, nemusí být nijak komplexní. Na základě toho, jsem navrhl jednoduché uživatelské rozhraní, z kterých komponent by se měla stránka skládat, přičemž jsem využil některé komponenty z knihovny React-Alantic, která byla mými kolegy vyvíjena.

5.1.3 Implementace

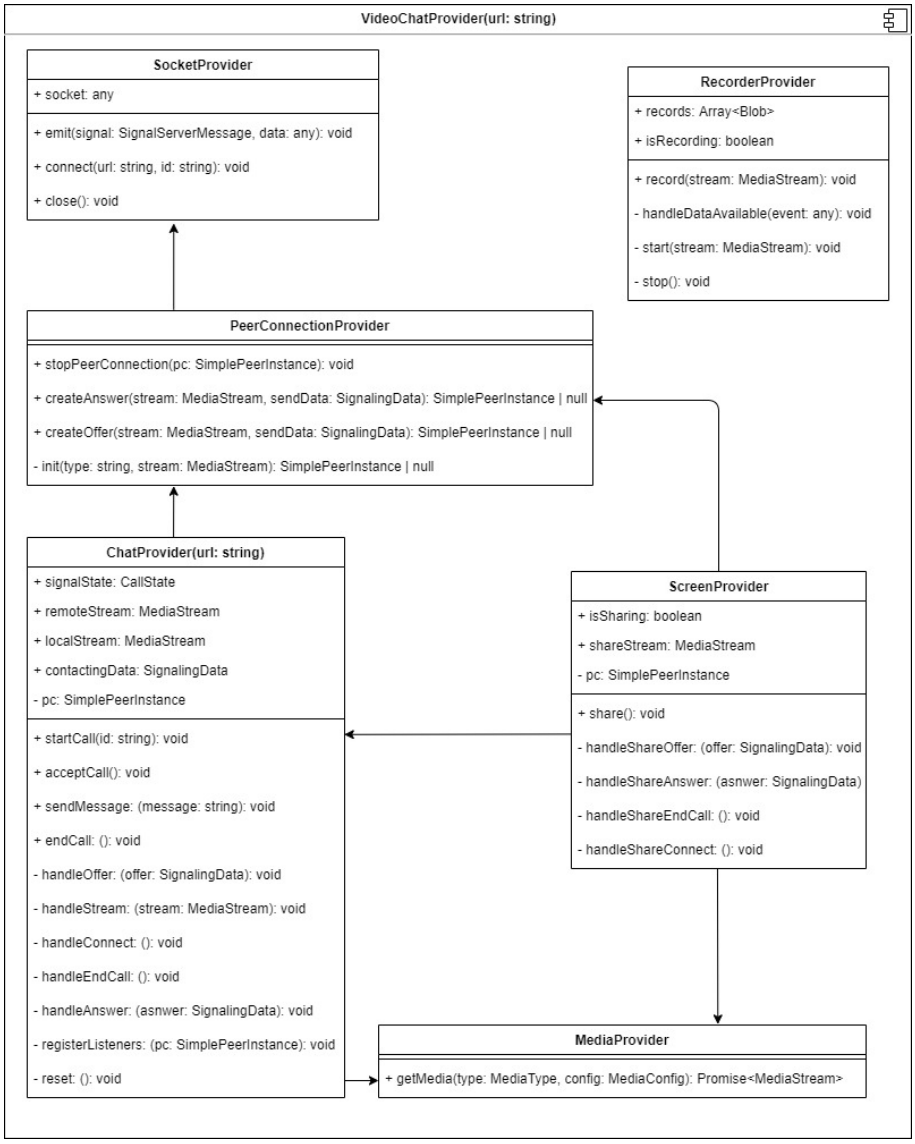
Dříve než jsem začal se samotnou implementací frontendové části projektu, jsem musel vytvořit signalizační server. K němu jsem využil *Node.js* [15], běhové prostředí, které umožňuje spouštět JS mimo webový prohlížeč a knihovnu *Socket.IO*, popsanou výše v kapitole 3.4 na stránce 15.

Signalizační server byl spouštěn na vyžádaném síťovém portu, kde po celou dobu běhu naslouchal na různé akce uživatelů. Uživatel při přihlášení okamžitě vytvořil spojení se serverem přes síťový soket. Server si informace o připojeném uživateli uložil do paměti.



Obrázek 3: Zjednodušený sekvenční diagram průběhu komunikace

Server následně spravoval spojení mezi dvěma klienty. Během běhu aplikace klient poslal na server signál o vytvoření hovoru se specifikovaným *id* druhého klienta a informacemi nutnými pro vytvoření P2P spojení, pak server, pokud měl informaci o druhém klientovi, druhého klienta kontaktoval. Po správné výměně signalizačních informací bylo mezi klienty vytvořeno P2P spojení. Tato linka byla ukončena přenosem signalizační zprávy s ukončením hovoru. Pokud spojení dvou klientů nemohlo být provedeno, pak server klientům poslal signál s chybovou hláškou.



Obrázek 4: UML diagram popisující strukturu logické části video chatu

Frontendovou stránku projektu jsem implementoval s závislostmi na modulu *simple-peer*, jednoduchým rozhraním nad technologií WebRTC, a *Socket.IO*, stejně jako u backendu.

Logickou část, která byla spojena s technologií WebRTC a komunikací přes síťový soket, jsem zapouzdřil do React rozhraní `Context`. Tuto implementaci zachycuje UML diagram na obrázku č. 4. Funkcionalita byla zapouzdřena a jednoduše využitelná v různých projektech. V závislosti na sobě a signálech obdržených ze serveru, funkcionální komponenty měnily svůj stav a poskytovaly metody, kterými jde hovor inicializovat, ukončit nebo přijmout.

Komponentou *VideoChatProvider* jsem obalil uživatelské rozhraní, implementované v komponentě *VideoChat*. Mimo jiné jsem zde využil poprvé modul React-Atlantic, z kterého jsem použil už nastylované komponenty: *Button*, *Title*, *Paragraph*, *Text*. Jedinou komponentu, kterou

jsem navíc k uživatelskému rozhraní zhotovil se nazývala *Video*. V ní jsem řešil kaskádové styly pro vizualizaci videa a problém zobrazení řešený pomocí podmíněného vykreslování v závislosti na dostupném datovém toku dat. Finálně dokončenou aplikaci zachycuje obrázek č. 24 (stránka 42, příloha A).

5.2 Úprava záložky - Titulní stránka

Zadáním byla kompletní úprava záložky Titulní stránka s následujícími požadavky:

- Maximální využití interně vyvíjené knihovny komponent React-Atlantic.
- Využití moderních principů knihovny React, což obnášelo obměnu starých nepřehledných třídních komponent za nové funkcionální komponenty.
- Zachování původní funkcionality, zároveň s rozšířením některých funkčních celků.
- Dodržení všech grafických prototypu.
- Integraci zcela responzivního designu.
- Vytvoření znovupoužitelných, co nejvíce nezávislých celků, tak aby bloky zdrojového kódu popisující logickou část i část uživatelského rozhraní mohly být konzumovány v celém systému znova, čímž zajistíme vyhnutí se duplikování zdrojového kódu.
- Uspořádání zdrojového kódu závislého na technologii Redux do jednotných částí, které budou moci býti jednoduše nahrazeny technologii Apollo Client a React rozhraním, `Context`.
- Otestování všech nově vytvořených celků.

5.2.1 Analýza

Moderní aplikace jsou v současné době implementované v souladu s kompletní responzivitou. Různé rozprostření elementů v rámci webové stránky je klasicky řešeno pomocí *CSS Media Queries*. Úpravou aplikace Codexis měly být *Media Queries* naprosto nahrazeny pomocí React rozhraní `Context`, pojmenovanou *Device*, kterou měl být responzivní design kontrolován skrz JavaScript.

Uživatel má v aplikacích možnost volby mezi tmavým a světlým tématem. Současně s úpravou bylo nutné tuto funkcionalitu včlenit do aplikace Codexis. Bylo nutné zhotovit `Context` *Theme*, prostřednictvím kterého byl sdílen objekt s nadefinovanými barvy pro různé téma aplikace. Zatímco sdílené barvy měly být využity ke stylování všech komponent.

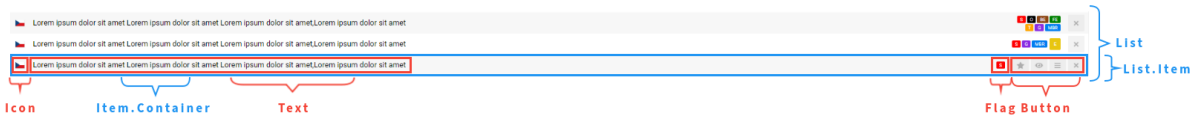
K stylování aplikace Codexis před úpravou byla využita technologie Less. Po předchozích zkušenostech však bylo rozhodnuto, že při úpravě je nutné odstranit kontrolu nad styly klasickým kaskádovým stylováním. Místo toho je nutná integrace CSS v JS. Zde měla být využita knihovna *Styled Components*, více popsána v kapitole 3.5 na stránce 15.

Automatické testy byly psány za pomocí technologie Cucumber. Testy pomocí kaskádových tříd vybíraly webové prvky na stránce, a tím kontrolovaly, zda jsou v webové stránce zahrnuty.

5.2.2 Návrh

Na základě poskytnutých návrhů uživatelského rozhraní bylo nutné promyslet, z jakých komponent se záložka Titulní stránka bude skládat. Celkově záložka byla rozvrhnutá do 5 větších komponent označených modře na obrázku č. 25. Současně lze na obrázku vidět funkční analýzu s dalšími využitými komponentami.

Ke každému uživateli se váže určitá historie vyhledávání, sledované nebo oblíbené dokumenty. K zobrazení těchto dat bylo nutné vytvořit seznam, v kterém bude zobrazena celá kolekce dat. Jeden řádek seznamu jsem v rámci zajištění principu znovupoužitelnosti rozdělil do více komponent. Situaci znázorňuje obrázek 5, na kterém jsou červeně označeny komponenty, které měly být využity z knihovny komponent React-Atlantic.



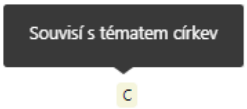
Obrázek 5: Návrh komponenty pro zobrazení dokumentů

5.2.3 Implementace

Implementaci jsem rozdělil do dvou celků. V prvním jsem vyřešil kaskádové stylování komponentou *List* a *List.Item*. *List* jsem implementoval jako bezstavovou komponentu, v které byly nadefinovány styly pro seznam. Zároveň jsem v ní řešil filtraci dětí, kdy průchodem všech dceřiných prvků jsem za pomocí vlastnosti *displayName* odstranil ty, které neodpovídaly přepokládané vlastnosti *displayName*. Komponenta *List.Item* byla stavová, z důvodu, aby si mohla uchovávat informaci o aktivním, vybraném prvku a další.

V dalším kroku jsem vytvořil komponentu *Item.Container*. V ní jsem řešil zobrazení dat pro jeden řádek v seznamu. Z grafických návrhů jsem vytvořil rozhraní pro vstupní parametry, kterými byly nepovinně množina příznaků, tlačítek, ikona a povinně text. Důležitou vlastností byla také funkce, která byla vyvolána při kliknutí na řádek. V daném řádku je limitován prostor pro zobrazení textu, proto při najetí na text je zobrazena nápověda s celým textovým řetězcem vlastnosti *text*.

Každý dokument v aplikaci Codexis má svůj příznak, který specifikuje typ dokumentu. Pro tento příznak jsem vytvořil komponentu *Flag*, viditelná na obrázku č. 6. Tu jsem sestavil ze dvou dílů: *Badge* a *Tooltip* (komponenty importované z interně vyvíjené knihovny komponent). Při přejetí myší přes komponentu *Flag*, jsem zobrazoval nápovědu v podobě krátkého popisu tykajícího se typu dokumentu. Komponenta *Flag* obohacovala vlastnosti z komponenty *Badge* o vazbou na předefinovaný objekt popisující pro typ dokumentu (klíč) barvu (hodnotu).



Obrázek 6: Komponenta Flag

Jako poslední mi scházelo vytvoření obálky pro vytvořené komponenty *List*, *List.Item*, *Item.Container*. Tuto komponentu jsem pojmenoval *ListDocument*. V ní docházelo k přemapování dokumentů do struktury využívající zmíněné komponenty. Komponentu *ListDocument* jsem integroval do 4 částí viditelných v návrhu zobrazeném na obrázku č. 25, kdy v komponentě *InputSearch* byla komponenta *ListDocument* zobrazena v našeptávači zobrazující možnosti vyhledávání.

Výsledek Titulní stránky v světlém i tmavém tématu lze vidět na obrázku č. 27, starý design na obrázku č. 26 (příloha B).

5.3 Knihovna komponent

Úprava aplikace Codexis byla spojena s návrhem nového uživatelského rozhraní, což z pohledu vývoje prezentační vrstvy knihovnou React obnášelo rozčlenění do komponent, kde komponenty vytvořené pro staré uživatelské rozhraní měly být nahrazeny novými.

Po prozkoumání aktuálních trendů ve světě, předchozích zkušeností a analýzou vytvořených požadavků pro aplikaci Codexis, bylo rozhodnuto, pro vytvoření knihovny komponent. Pomocí vytvořené knihovny měly být redukovány problémy vyskytující se u předchozí implementace. Ta se potýkala s kolizí kaskádových stylů a nekonzistentním vývojem designu. Mimo jiné měla být tímto krokem zdokonalena spolupráce mezi frontend developery a UX designérem u vývoje produktů.

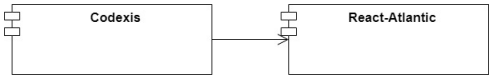
5.3.1 Specifikace požadavků

Cílem bylo vytvořit distribuovatelnou knihovnu komponent zapouzdřující designové prvky, atomy, molekuly a organismy. Výsledná kolekce komponent, měla respektovat množinu specifikovaných požadavků: znovupoužitelnost, škálovatelnost, abstraktnost, flexibilita, korektnost komponent, dokumentace.

5.3.2 Analýza a návrh

Knihovna měla být distribuována tak, aby ji bylo možné konzumovat v různých projektech. Zdrojový kód knihovny byl zapouzdřen v samostatném repositáři (více obrázek č. 7), což umožnilo v případě potřeby knihovnu přidat do závislostí vyvíjeného softwaru, pomocí správce balíčku (*yarn* [16], *npm* [17]).

Z popisu požadavků byly známé principy, s jakými mají být komponenty vyvíjeny. Zásadním krokem bylo navrhnutí rozhrání pro komponentu, popisující objekt, vlastnosti (props). U tohoto



Obrázek 7: Správa balíčku - React-Atlantic

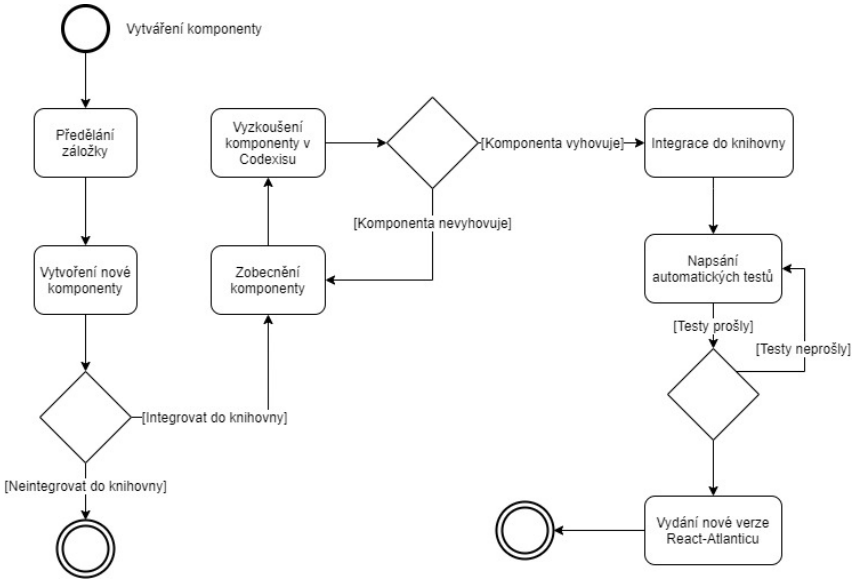
návrhu jsem se rozhodl hledat inspiraci, již u fungujících knihoven dostupných veřejně na internetu (React-Bootstrap [18], Material UI [19], Ant Design [20]), kde jejich vlastnosti jsem upravil podle potřeby na míru.

Zachytávání událostí v JavaScriptu je vyvoláno s jedním argumentem *event*. Z důvodu vyhnutí se stálého odchytávání tohoto argumentu, byly do knihovny zároveň integrovány metody bez argumentu, které poskytovaly rozhraní k těmto metodám. Kromě metod na odchytávání událostí, zde byly umístěny metody, které byly vyhodnoceny jako dostatečně abstraktní, logika na udržení responzivity a funkcionalita poskytující správu barevného tématu aplikace.

Aby byla knihovna dostatečně efektivní, bylo nezbytné všechny komponenty kvalitně popsat. K tomu byl využit nástroj *Storybook* [21] sloužící k organizovanému vývoji UI komponent. Tím byla zajištěna dokumentace všech komponent včetně prostředí, kde vytvořené komponenty můžeme vyzkoušet [22].

5.3.3 Metoda pracovního postupu vývoje komponent

Na začátku praxe byly komponenty vyvíjeny rovnou v modulu React-Atlantic, komponenta byla vytvořena, prošla si fázi testování, a poté byla vydána nová verze balíčku. V případě, že se našla chyba, musela být upravena v modulu a znova vydána nová verze. Tento postup nevyhovoval potřebám, a proto byl v průběhu upraven. Metoda nového postupu vývoje lze vidět na obrázku č. 8.



Obrázek 8: UML diagram popisující postup vývoje komponent

5.3.4 Skeleton

Často vzniká situace, kdy obsah určité komponenty ještě není připraven, v tu chvíli je použita komponenta *Skeleton*. Principy použití:

- Pokud komponenta obsahuje větší množství informací, například dlouhý seznam.
- V případě využití vzoru pro postupné načítání dat.
- Vytvoření dojmu čekání.
- Prvotního vykreslení obsahu.

Z pohledu odlišného uživatelského požitku, ale stejné funkčnosti bývá komponenta *Skeleton*, často nahrazena komponentou *Spiner*.

5.3.5 Analýza a návrh

Před implementační fází bylo nutné provést dostatečnou analýzu z pohledu grafických návrhů, více na obrázku č. 9.

Obrázek 9: Grafický návrh komponenty - Skeleton

Na základě designu jsem navrhl, jaké vlastnosti by měla komponenta přijímat. Dohromady bylo navrhnuto 7 vlastností splňující všechny potřeby. Více o vlastnostech na obrázku č. 28 (příloha C).

5.3.6 Implementace

Skeleton je bezstavový, což znamená, že je zcela řízen z nadřazené komponenty. K překreslení dochází v případě, že hodnota některého vstupního parametru je změněna.

V komponentě bylo potřeba vyřešit všechny problémy v kaskádových stylech, proto všechny vlastnosti byly předány v návratovém bloku stylované komponentě, která byla jediná vrácena.

Tvar nabýval hodnot obdélníku a kruhu. V případě kruhu jsem podmínečně přidal v kaskádových stylech vlastnost *border-radius* s hodnotou *100%*. Druhým důležitým problémem byla velikost, implementována tak, aby se přizpůsobovala velikosti nadřazeného prvku. Velikost mohla být specifikována pomocí definovaného typu *Size* nebo udána přesnými hodnotami šířky (*width*) a výšky (*height*). Pomocí takhle modifikovatelné velikosti mohla být komponenta vytvarována podle požadavků. Další tvary *Skeletonu* jsou zobrazeny na obrázku č. 10. V průběhu čekání na data probíhala animace, její barva nemusí být v některých barevných tématech viditelná, a proto musela být modifikovatelná skrz vlastnost *bgColor* a *animationColors*.

Obrázek 10: Modifikovatelná velikost skeletonu

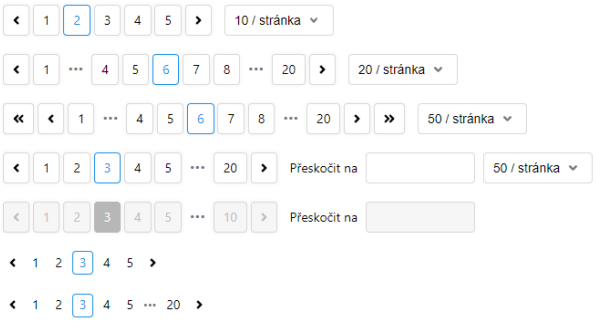
5.3.7 Pagination

Pagination, neboli česky stránkovač je komponenta, která umožňuje využít techniku stránkování. Využívá se:

- V případě, že uživatel chce procházet daty, může použít navigaci přes stránky.
- Použitím techniky stránkování dochází při větším počtu dat k omezení položek, čímž odpadá problém dlouhého načítání (vykreslování) dat.

5.3.8 Analýza a návrh

Po analýze grafických návrhů (obrázek č. 11) jsem vytvořil vlastnosti (obrázek č. 29, příloha C).



Obrázek 11: Návrh komponenty - Pagination

Důraz u vývoje knihovny komponent je kladen na použití již vytvořených částí. Nové komponenty lze snadno poskládat z již otestovaných, fungujících částí. Z toho důvodu bylo nutné prozkoumat všechny potencionálně použitelné komponenty. Výhodou je, že poté co je upravena jedna, je celá úprava promítnuta do všech ostatních.

5.3.9 Implementace

Stránkovač jsem implementoval jako stavovou komponentu, která ve svém stavu uchovává informace o zvolené stránce a zvoleném počtu položek pro stránku. Zároveň bylo nutné implementovat rozhraní, kde uživatel může vstupními parametry ovlivnit interní stav. Naslouchání na změnu vlastnosti *pageSize* a *current*, použitím rozhraní `useEffect`, jsem hodnoty vstupních parametrů synchronizoval se stavem.

První část, z které se stránkovač skládal byla *ButtonList*. V něm jsem vykreslil posloupnost čísel, využitím již vytvořené komponenty *Button* a *Icon*. Vykreslení této posloupnosti jsem rozdělil na dva scénáře, ty jsou mimo jiné zobrazeny na obrázku č. 12:

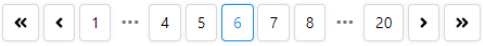
- **middleScenarion** - Případ, kdy vzdálenost aktuálně vybrané stránky od okrajů posloupnosti je větší nebo rovna tří. V tom případě jsou vykresleny tlačítka, kolem aktuálně vybrané stránky ve vzdálenosti 2, zároveň první a poslední tlačítko.
- **noMiddleScenarion** - Vykreslila od první nebo poslední stránky čtyři tlačítka. Zde byly vyřešeny ostatní speciální případy (prázdný stránkovač, vykreslení pouze jedné stránky nebo vykreslení menšího počtu tlačítek).

(a) middleScenario (b) noMiddleScenario

Obrázek 12: Verze posloupnosti tlačítek v stránkovači

Při kliknutí na číslo stránky byla zavolána funkce pro změnu stránky, tím dochází v rodičovské komponentě k získání dat v podobě čísla reprezentující kliknutou stránku a aktuálně vybrané velikosti stránky.

Kromě tlačítek reprezentující stránku bylo v posloupnosti možno vykreslit speciální tlačítka, která umožňovala přeskakovat o jednu stránku v určeném směru, přeskočit na konec nebo začátek posloupnosti, nebo přeskočit o specifikovaný počet stránek v daném směru. Každou aktualizaci stránky provedenou pomocí těchto metod jsem musel limitovat hranicemi posloupnosti, jinak by mohlo dojít k překročení hranic posloupnosti. Zmíněné funkce byly zapouzdřeny v bezstavových komponentách *ArrowButton* a *ThreeDots*, viditelné na obrázku č. 13. Z nich *ArrowButton* byla bezstavová komponenta a *ThreeDots* stavová, kde ve stavu byla uchována informace o tom, jestli má uživatel na komponentě kurzor a informace o kliku na tlačítku. Tyto informace mi sloužili k rozhodnutí, zda má být vykreslená nápověda a jestli má být viditelná komponenta s funkcí krokového přeskoku na stránku. Všechno toto rozhraní k stránkovači mohlo být skryto v závislosti na *boolean* informaci získané ve vstupních parametrech.



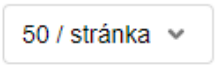
Obrázek 13: Vzhled subkomponenty ButtonList [22]

V komponentě *QuickJumper* jsem implementoval uživatelské rozhraní, prostřednictvím kterého mohl uživatel přeskočit na specifikovanou stránku. Zadaná hodnota byla udržována ve stavu.

Poslední částí byla bezstavová komponenta *SizeChanger*, kterou uživatel mohl změnit počet položek na stránce. Možnosti, které uživateli byly k dispozici, komponenta přijala jako kolekci

Obrázek 14: Vzhled subkomponenty QuickJumper [22]

čísel. Změnou vybrané možnosti byla vyvolána akce pro změnu velikosti stránky, a tím dochází v rodičovské komponentě k získání dat stejně jako u změny zvolené stránky.



Obrázek 15: Vzhled subkomponenty SizeChanger [22]

Většina vstupních parametrů s typem *boolean* dovolovala skrývat funkcionální celky s určitým rozhraním, kromě toho, proměnná *isSimple* poskytovala uživateli schopnost vykreslení stránkovače v jednoduché verzi.

Obrázek 16: Vzhled jednoduché verze stránkovače [22]

5.3.10 Carousel

Komponenta *Carousel* je známá pod případem užití, kdy jsou uživateli prezentovány fotky, obrázky. Obsah elementů je umístěn v *kartách*, mezi kterými může uživatel přepínat. Obecně se využívá:

- Potřebujeme vykreslit více obsahu na stejné úrovni.
- Nemáme dostatek místa na vykreslení obsahu.
- Chceme ušetřit místo na stránce.

5.3.11 Návrh

V rámci této komponenty nebyly k dispozici žádné grafické návrhy, jelikož nejde ani tak o implementaci vizuální stránky, ale o stránku logickou. Z popisu bylo jasné, že obsah komponenty si bude moci uživatel vybírat, proto se bude dělit na část, která lze vidět, a tu, která je schovaná.

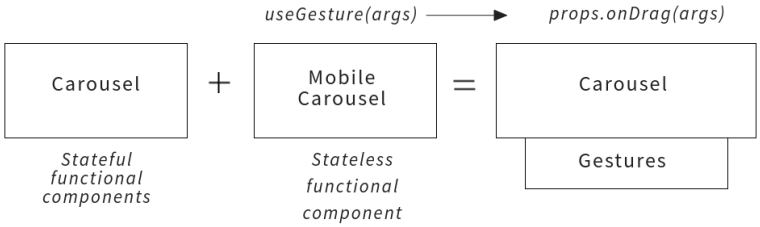
Vyžádáním změny viditelného obsahu dojde ke změně stavu dvou prvků na sobě korelujících. Oba prvky, závisle na sobě negují svůj stav z neviditelného na viditelný, nebo naopak. Bylo nutné, aby uživatel zůstal informovaný o současném stavu obou prvků, proto přechod prvků musel být plynulý, neboli animovaný.

Z tohoto důvodu bylo nezbytné zanalyzovat jakým způsobem bude animace prvku kontrolována. Vzhledem k užití React-Spring, popsaný výše v kapitole 3.9 na stránce 16, v aplikaci

Codexis, jsem pro kontrolu animace zvolil stejnou knihovnu. Tím nebylo nutné přidávat novou závislost do projektu.

Chování jsem rozdělil na dva případy užití podle použitého zařízení. Do první skupiny byly zařazeny zařízení používající dotykovou technologii (mobilní zařízení, tablet) a do skupiny druhé spadal zbytek (stolní počítač, laptop). Důvodem rozdělení byl odlišný způsob ovládání, kde druhá skupina v porovnání s první postrádá ovládání pomocí *gest*. K implementaci metod poskytující rozhraní *gest*, jsem použil knihovnu *React-Use-Gesture* (viz kapitola 3.10 na stránce 16).

Z tohoto důvodu, jsem se rozhodl použít techniku HoC (*Higher-Order Component*). Tato kompoziční metoda je znázorněna na obrázku č. 17.



Obrázek 17: Diagram s technikou Higher-Order Component

Vytvořené vlastnosti komponenty, s typy a krátkým popisem, jsou zobrazeny na obrázku č. 30 (příloha C).

5.3.12 Implementace

Uvnitř jsem udržoval informaci o všech vyfiltrovaných dětech splňující kritérium správné subkomponenty *Carousel.Slide*, šířku nadřazeného elementu, který plnil funkci kontejneru, výšku obsahu aktuální *karty*, a číslo, které indikovalo aktuálně vybranou *kartu*. Stejně jako u předchozích komponent je možné kontrolovat externě, ale chování musí být fungující i bez externí kontroly, z toho důvodu jsem původní stav nastavoval proměnnými *default*, a registroval posluchače skrz metodu `useEffect`.

Změnou aktuálně vybraného elementu došlo k vyvolání funkce, s daty aktuálně vybraného elementu. Současně došlo k nastavení nového stavu, ve kterém jsem před nastavením volal metodu `clamp` vracející číslo v rozmezí velikosti počtu elementů.

V množině stavů byla navíc ještě uchována informace o animovaných stylech pro všechny elementy nazvané *springs*. Pomocí nich jsem měnil proměnné *offset*, *opacity* a *touchAction*. Ve výsledku byl pro element animován rovnoměrný nepřímočarý pohyb po *horizontální* ose (x).

Rozhodl jsem se obsah neomezovat žádným počtem elementů. Abych předešel problémům s výkonem, byla přidána do animovaných stylů vlastnost *display*. Pokud vzdálenost elementu

od aktuálně vybraného elementu byla větší jak dva, pak kaskádová vlastnost nabývala hodnoty *none*, k tomu nebylo nutné vykreslovat obsah element.

Pro dotykové zařízení jsem implementoval gesta. Skrz rozhraní `useDrag`, jsem obdržel informaci o pohybu prstu na displeji, souřadnicích prvního doteku, posledního doteku, čase kdy uživatel začal gesto a rychlosti. Podřízeně na těchto proměnných, a šířce kontejneru jsem nastavoval styly a aktuálně vybranou kartu.

5.4 Úprava záložky - Vyhledávání

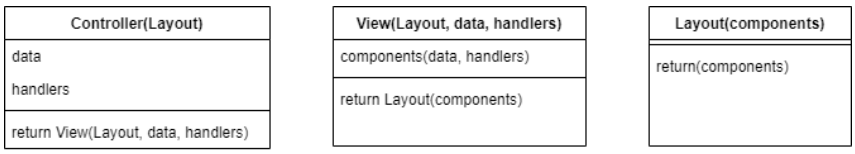
Podobně jako u záložky Titulní stránka bylo nutné u záložky Vyhledávání opět dodržet požadavky popsané v kapitole 5.2 na stránce 23.

V rámci některých nově vytvářených komponent byl duplikován zdrojový kód kvůli neekvivalentní byznys logice, různého pořadí elementů, nebo jiné velikosti tlačítka v jistém responzivním rozlišení. Podobné komponenty tak nemohly být znovupoužity a docházelo k porušení několika předem definovaných požadavků.

U systému Codexis se nejedná o jednoúčelnou aplikaci, jsou zde předpokládaný nové požadavky na změny, obecně škálovatelnost aplikace, a proto bylo nutné navrhnout architekturu takovou, která tyto principy umožní dodržet.

5.4.1 Analýza a návrh

Před začátkem implementace záložky bylo potřeba vyřešit strukturování tak, aby mohla být stejná vizuální komponenta využita s odlišnou logikou a jiným rozvržením na stránce. Návrh byl diskutován skrz celý frontendový tým, kde po diskuzi bylo vymyšleno řešení s rozvrstvením komplexních komponent do třech vrstev. Vývoj je pak dostatečně intuitivní, známý z softwarového inženýrství, kde komplexnější komponenta není řešena celistvě v jedné části (vrstvě), ale je využito více vzájemně spolupracujících vrstev, které si zpravidla mezi sebou vyměňují potřebná data. Obrázek č. 18 zobrazuje navrhlou třívrstvou architekturu, kde mimo jiné lze vidět pojmenování jednotlivých vrstev. Ke každému dílčímu celku komponenty (vrstvy) byl pak k názvu komponenty přidán, jako sufixová část slova, název vrstvy.



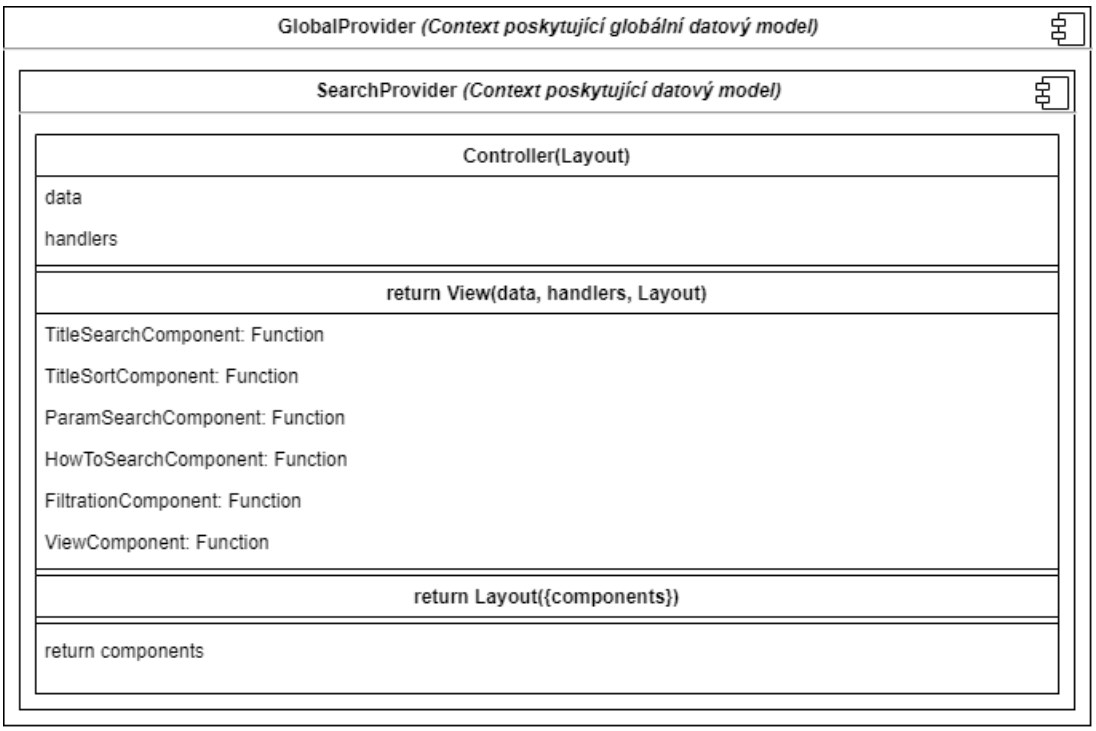
Obrázek 18: Architektura Layout-View-Controller

Stejně jako u záložky Titulní stránky reimplementace byla podložena seznamem návrhů uživatelského prostředí pro všechna responzivní rozlišení. Na základě těchto návrhů, které byly vytvořené designérem specializujícím se na interakce uživatele s produktem, bylo navrhnuto, z jakých komponent by se záložka Vyhledávání měla skládat. Výsledek segmentace pro deskto-

pové rozlišení je ilustrován na obrázku 31 (strana 47, příloha D). Celkově byla stránka rozčleněna do třech rozměrnějších bloků, z toho jeden jsem měl na starost já (označen červeně).

5.4.2 Implementace

Při vytváření komponenty *Widget.Search* jsem dbal na dodržení třívrstvé architektury, proto jsem komponentu segmentoval do tří funkcionálních komponent, jedné stavové (*WidgetSearchController*) a dvou bezstavových (*WidgetSearchView*, *WidgetSearchLayout*). Výslednou implementaci ilustruje obrázek č. 19.



Obrázek 19: UML diagram zobrazující implementaci komponenty Widget.Search

První vrstvu jsem implementoval jako stavovou funkcionální komponentu. Napojil jsem ji na vytvořený `Context`, z kterého čerpala datový model. Nad datovým modelem byl proveden výpočet, transformace, kde po zpracování došlo k předání všech dat do následující vrstvy. Současně zde byly nadefinovány metody, které byly spouštěny na vyžádaní uživatele. Tato komponenta (*Controller*) je tedy jednotkou, která se stará o provázání funkčnosti mezi datovým model a vizuální reprezentací.

K přijatým datům a metodám jsem ve druhé vrstvě vytvořil vizuální reprezentaci, kterou jsem umístil do funkcí a konstant. V návratovém bloku komponenty byly všechny části předány jako proměnné do poslední vrstvy, která se starala o finální vykreslení. Dále jsem v této vrstvě řešil výkonnostní optimalizace. Pomocí rozhraní `useMemo` jsem jednotlivé části udělal závislé na

určitých datech, kde po aktualizaci přijatých dat nebo metod pak docházelo k překreslení pouze částí, které byly na aktualizovaných proměnných závislé. Ostatní části nebylo nutné překreslovat.

Argumentem bezstavové komponenty umístěné na poslední vrstvě (*Layout*) byl objekt s množinou všech částí, které nadefinovala předchozí vrstva. Podstatou této funkce bylo dynamické vykreslení jednotlivých částí ve vyžádaném rozložení.

5.4.3 Shrnutí

Díky využití třívrstvé architektury jsem sloučil *Widget.Search* s komponentou, která byla užita v první upravované záložce Titulní stránka. Z této staré komponenty jsem separoval byznys logiku a grafické rozvržení v rámci všech responzivních rozlišení. Pro záložky byly implementovány odlišné vrstvy definující kontrolu a rozložení, kde vrstva popisující vizuální reprezentaci zůstala stejná. Tentýž přístup byl aplikován i v dalších záložkách.

Úkolů, které jsem prováděl na této záložce, bylo více. Pouze jsem se zmínil o *Widget.Search*, jelikož tento problém byl ze všech problémů nejrozsáhlejší.

Finální výsledek předělané záložky i se starým uživatelským rozhraním je možné vidět na obrázku č. 33 a č. 32 (příloha D).

5.5 Widget prezentující novinky a aktuality

Firma Atlas Group je provozovatelem právního portálu s názvem *Právní prostor* [23]. Tento portál nenavštěvují denně jenom lidé z oboru práv, ale i lidé, kteří hledají právní informace.

Obsah portálu je tvořen komentáři k různým změnám v legislativě, články k aktuálním tématům, a hlavně současnými novinkami.

Aplikaci Codexis používá část čtenářů *Právního prostoru*. Proto by bylo obrovským bonusem těmto uživatelům nabídnout pohodlný přístup k publikacím, které by byly přehledně prezentovány přímo v aplikaci Codexis.

5.5.1 Specifikace požadavků

Datová část k publikacím byla předpřipravena backend týmem. Mým zadaným cílem v této úloze bylo vytvořit prezentační vrstvu, prostřednictvím které budou uživatelům navštěvující aplikaci Codexis prezentovány novinky a aktuality dostupné skrz předpřipraveného zdroje dat.

V požadavcích byla upřesněna technologie Apollo, která měla být použita souběžně s nástrojem Codegen při vývoji.

5.5.2 Analýza a návrh

Pro požadovanou prezentační vrstvu byly opět, jak v předchozích případech, vytvořeny grafické návrhy (viz obrázek č. 34). Po analýze grafických námětů jsem navrhl komponentu *Carousel*, nezbytnou pro prezentaci novinek na mobilním a tabletovém zařízení, kterou jsem popsal v podkapitole č. 5.3.10 na stránce 30.

Před vytvořením prezentační vrstvy bylo nutné získat data z datové vrstvy aplikace Codexis. Datová vrstva využívá technologii GraphQL, která je popsána v technologickém řešení v kapitole 3.6 na stránce 16. Byl požadavek na integraci Apollo klienta, proto bylo nutné knihovnu zanalyzovat a přidat potřebné závislosti do projektu.

Následným požadavkem bylo využití nástroje Codegen. Pomocí technologie GraphQL je k API dostupná celá dokumentace popisující dostupné dotazy a datové typy. Podstatou využití tohoto nástroje pak je, že dokáže ze schématu vygenerovat zdrojový kód pro nadefinované datové typy. Zároveň správnou konfigurací souboru *codegen.yml*, dokáže nástroj vygenerovat funkce, prostřednictvím kterých můžeme dotazovat backend služby.

Posledním provedeným krokem v rámci této analýzy bylo prozkoumání aplikačního rozhraní, kterým byly zpřístupněny publikace. V metodě byly obsaženy dva vstupní parametry. *Limit* specifikující počet žádaných publikací, a proměnná *offset*, kterou byl reprezentován krok od první dostupné publikace v rámci celé kolekce.

5.5.3 Konfigurace aplikace

Abych měl k dispozici aktuální API, bylo nutné stáhnout nové GraphQL schéma. Po stažení došlo k aktualizaci souboru *schema.graphql*, kdy v něm přibyl nový datový typ *newsFeedItem* reprezentující jednu publikaci a metoda `newsFeed`.

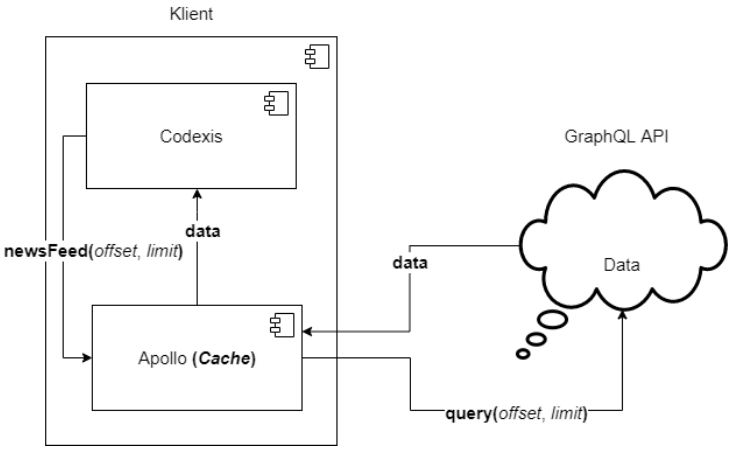
Pro tuto metodu bylo nutné vytvořit do adresáře *query* nový dotaz (*newsFeed.graphql*). V tomto dotazu jsem volal metodu `newsFeed`, a díky GraphQL, jsem mohl vybrat potřebná data, která chci v rámci jedné publikace vrátit.

Poté, co jsem měl připraven dotaz a stáhnuté schéma, jsem upravil soubor *codegen.yml*, který byl nakonfigurován tak, aby vygeneroval současně s datovými typy i metody dotazující datovou vrstvu.

Po nakonfigurování všech věcí jsem před vytvářením komponent musel obalit celou aplikaci *ApolloProviderem*, kterému byla předána do parametrů proměnná *client*, předem vytvořená s parametry *cache* a *link*, sdílená napříč celé aplikace, bez nutnosti manuálního předávání. Díky něj bylo možné využít v komponentách o úroveň níže rozhraní pro získání dat z knihovny Apollo Client. Rozšíření aplikace o Apollo klienta s průběhem získání dat z backend služeb je možné vidět na obrázku č. 20.

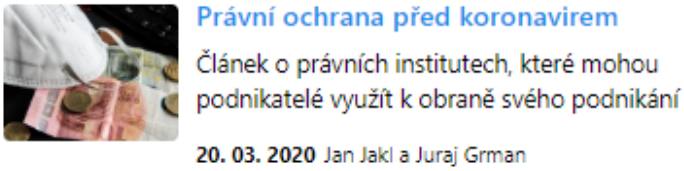
5.5.4 Tvorba komponent

Prezentace novinek měla být integrována do záložky Titulní stránka. Rozhodl jsem se vytvořit komponentu *Widget.News* způsobem popsaným v podkapitole 5.4.2 na stránce 33. V ní jsem využil vygenerovanou metodu `useNewsFeed`, kterou jsem později zapouzdřil do vytvořeného rozhraní, React `Hook`, *funkce prostřednictvím které lze ve funkcionální komponentě používat stav, životní cyklus a ostatní věci*, rozšířeného o postupné načítaní popsané níže (v podkapitole 5.5.5).



Obrázek 20: Apollo klient

Poté, co jsem měl připravené data, jsem musel vytvořit komponentu pro prezentaci jedné publikace *NewsItemContainer*. Tu jsem vytvořil jako bezstavovou komponentu, v niž jsem vyřešil dva druhy responzivního rozlišení, desktopovou a mobilní verzi. Mimo jiné bylo nutné zobrazení textu v určitých rozlišeních omezit na 150 znaků. Komponentu lze vidět na obrázku č. 21.



Obrázek 21: Komponenta pro jednu publikaci novinek

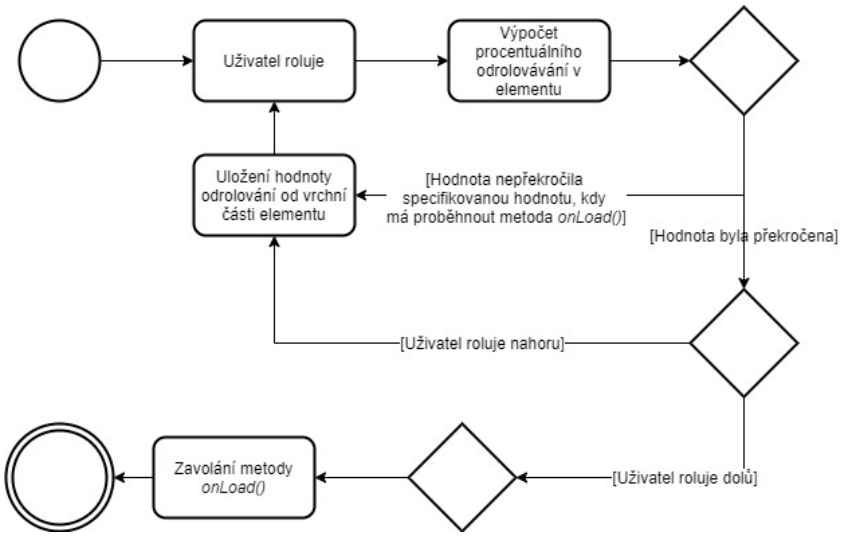
Následně jsem vytvořil komponentu *NewsList* pokrývající seznam publikací s využitím komponenty *NewsItemContainer*. Zároveň bylo nutné vytvořit vizualizaci publikace, která načítá (viz obrázek č. 22)

Obrázek 22: Komponenta zobrazující načítání publikace

Pro prezentaci novinek měla být vytvořena nová záložka Novinky a aktuality. V ní jsem využil již zmíněné komponenty. Akorát bylo nutné publikace rozdělit podle zveřejněného datumu. Proto jsem využil datovou strukturu slovníku, kde jsem data připravil ve formátu datum (klíč), seznam publikací (hodnota).

5.5.5 Implementace postupného načítání dat

Funkčnost postupného načítaní jsem zapouzdřil do rozhraní, React `Hook`, s názvem `useInfiniteScroll`. Vstupními parametry rozhraní byl *rolovací element*, *číslo*, reprezentující procentuální odrolování od vrchní části rolujícího kontejneru a *metoda*, popsána níže v tomto odstavci. V době, kdy je komponenta již vložena v objektovém modelu dokumentu, se na vstupní parametr *element*, registroval posluchač, který kontroloval procentuální rolování uživatele. Pokud procentuální hodnota odrolování překročila hodnotu *when*, pak byla zavolána metoda `onLoad` (viz obrázek č. 23) jejímž výstupem byly další načtené publikace.



Obrázek 23: UML reprezentující postupné načítaní dat

Zatímco v rolovacím kontejneru byly novinky postupně načítány v závislosti na procentuálním odrolování, v komponentě *Carousel* metoda `onLoad` volána v době, kdy aktuální *karta* byla od poslední vzdálená o vzdálenost 1.

Výsledek zachycuje obrázek 36 a 35 (strana 50, příloha E).

6 Shrnutí

Mé očekávání výběrem praxe se splnilo, absolvování pro mě bylo obrovským přínosem, z kterého jsem získal spoustu praktických zkušeností. Kromě praktických zkušeností jsem spolupracoval se skupinou lidí, díky nichž jsem měl možnost se rychleji rozvíjet jak v programátorských schopnostech, tak i těch komunikačních.

Rád bych zmínil také agilní přístup vývoje softwaru. Poznal jsem, že každá fáze je u vytváření kvalitního produktu důležitá. Hlavně fáze návrhu a analýzy, často opomíjená fáze, u které se z mého pohledu jedná o nejdůležitější dílčí celek. Mým poznatkem je, že díky dobrého návrhu se vyhneme v pozdějších fázích vývoje nemilým komplikacím.

Ve spojení s poznamenaným návrhem, bych rád napsal krátký odstavec o abstraktnosti. Je důležité, aby tvorba aplikace byla vyvíjena s ohledem na určitou míru obecnosti, ne jako jeden celek, ale spíše jako více menších částí. Tím jsme schopni zajistit vlastnost znovupoužitelnosti. Úprava funkcionality nebo funkcionalita nová pak není problém, díky využití již vytvořeného bloku kódu.

6.1 Uplatněné znalosti

Mezi množinu uplatněných znalostí nabytých na akademické půdě bylo mnoho, především z předmětů zabývající se webovými technologiemi. Využil jsem znalosti programovacího jazyka JavaScript z Vývoje mobilních aplikací I nebo předmětu Vývoje internetových aplikací, kde se úzce zavadilo i o pokročilejší technologie, spíše teoretickou formou, TypeSript, React a další.

Většinou bývá pravidlem, že syntax se často opakuje napříč spousty programovacích jazyků. Naučením jednoho programovacího jazyka je pak o to jednodušší naučení dalšího. V tom mi pomohly předměty jako Programovací jazyky I a II. Mimo syntaktickou stránku bych chtěl poznamenat řešení algoritmických úloh, naučených v předmětech Algoritmy nebo Úvodu do teoretické informatiky. Vyřešení algoritmu *Prohledávání do hloubky* jsem čelil například i při přijímacím pohovoru. Během praxe jsem také čelil mnohdy správnému návrhu, předmět Vývoj informačních systémů mě donutil využít schopnost navrhnout správnou architekturu.

6.2 Scházející znalosti

Ačkoliv jsem nabyl spoustu znalostí na akademické půdě, spousta mi jich scházelo, především pak těch praktických.

Při nástupu mi scházelo hned několik znalostí, a to jak schopnost verzování projektu využitím technologie Git, tak i znalost knihovny React a technologií s ní spojenou. Z počátku praxe jsem všechny tyto neznalosti musel dohnat, a to se odrazilo na čase stráveném při řešení úkolů. Po určité době jsem již byl schopen neznámé technologie ovládat na dobré úrovni. Z toho jsem si také odnesl, že než podrobná znalost technologie, je důležitější znalost postupů, kterými se technologie řídí.

7 Závěr

Svou činností na praxi jsem pomohl s rozvojem aplikace Codexis. Splněním zadaných úkolů jsem se zúčastnil kompletního přepsání aplikace a zároveň jsem některými úkoly rozšířil stávají funkčnost. Tímto jsem měl možnost se účastnit na vývoji reálného softwaru, který je používán v řádu tisíců lidí denně, a to od návrhu, přes implementaci, až po testování a nasazení do produkce.

Díky možnosti absolvovat praxi ve firmě Atlas consulting spol. s r.o jsem nabyl nespočet zkušeností, které můžu zúročit v profesním životě.

Mezi získané zkušenosti bych především zmínil knihovnu React a s ní knihovny spojené, na kterých aplikace Codexis stojí. Kromě specifických technologií, jsem měl možnost se rozvíjet v obecných principech programování, schopnosti řešit přidělený problém a schopnosti naučení neznámé technologie.

Nejedenkrát jsem se potýkal s řešením více než jednoho úkolu, díky toho jsem se naučil správně organizovat svou práci.

Literatura

1. *ATLAS consulting* [online]. Ostrava: ATLAS consulting, c2013-2019 [cit. 2020-02-20]. Dostupné z: https://atlasconsulting.cz/.
2. *Codexis: Komplexní národní a evropský právní informační systém* [online]. Ostrava: ATLAS consulting, c2013-2019 [cit. 2020-04-19]. Dostupné z: https://atlasconsulting.cz/software/codexis/.
3. *TypeScript* [online]. Redmond, Boston, SF & NYC: Microsoft, c2012-2020 [cit. 2020-02-25]. Dostupné z: https://www.typescriptlang.org/.
4. *React: A JavaScript library for building user interfaces* [online]. Facebook, c2020 [cit. 2020-02-25]. Dostupné z: https://reactjs.org/.
5. *WebRTC: Real-time communication for the web* [online]. Google [cit. 2020-02-25]. Dostupné z: https://webrtc.org/.
6. *WebRTC in the real world: STUN, TURN and signaling* [online]. Sam Dutton, 2013 [cit. 2019-02-25]. Dostupné z: https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/.
7. *Socket.io* [online] [cit. 2020-03-01]. Dostupné z: https://socket.io/.
8. *Styled components* [online]. Glenmaddern [cit. 2020-03-04]. Dostupné z: https://styled-components.com/.
9. *CSS vs. JavaScript: Trust vs. Control* [online]. Christian Heilmann, 2017 [cit. 2019-03-04]. Dostupné z: https://christianheilmann.com/2017/06/21/css-vs-javascript-trust-vs-control/.
10. *GraphQL* [online]. The GraphQL Foundation, c2020 [cit. 2020-03-03]. Dostupné z: https://graphql.org/.
11. *Apollo* [online]. Meteor Development Group, c2020 [cit. 2020-03-03]. Dostupné z: https://www.apollographql.com/docs/react/.
12. *GraphQL code generator* [online]. The Guild, c2020 [cit. 2020-03-14]. Dostupné z: https://graphql-code-generator.com/docs/plugins/typescript-react-apollo/.
13. *React-spring* [online] [cit. 2020-03-04]. Dostupné z: https://www.react-spring.io/.
14. *React Use Gesture* [online]. React Spring, c2020 [cit. 2020-03-04]. Dostupné z: https://use-gesture.netlify.app/.
15. *Node.js* [online]. OpenJS Foundation [cit. 2020-03-01]. Dostupné z: https://nodejs.org/en/about/.
16. *Yarn: Safe, stable, reproducible projects* [online] [cit. 2019-03-16]. Dostupné z: https://yarnpkg.com/.

17. *Npm* [online]. Npm, 2014 [cit. 2019-03-16]. Dostupné z: https://www.npmjs.com/.

18. *React Bootstrap* [online] [cit. 2019-03-17]. Dostupné z: https://react-bootstrap.github.io/.

19. *MATERIAL-UI* [online]. Material-UI, c2020 [cit. 2019-03-17]. Dostupné z: https://material-ui.com/.

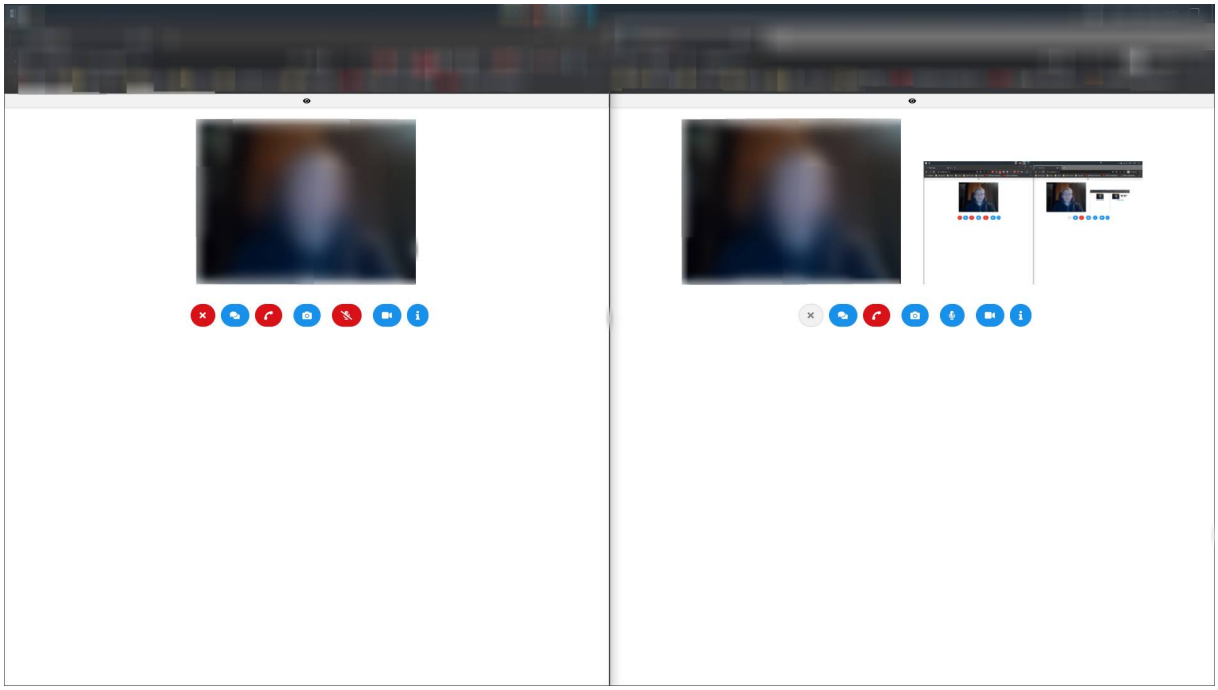
20. *Ant Design* [online]. XTech [cit. 2019-03-17]. Dostupné z: https://ant.design/.

21. *Storybook: Build bulletproof UI components faster* [online] [cit. 2020-03-04]. Dostupné z: https://storybook.js.org/.

22. *React Atlantic* [online]. thepatriczek [cit. 2019-03-04]. Dostupné z: https://thepatriczek.github.io/React-Atlantic/.

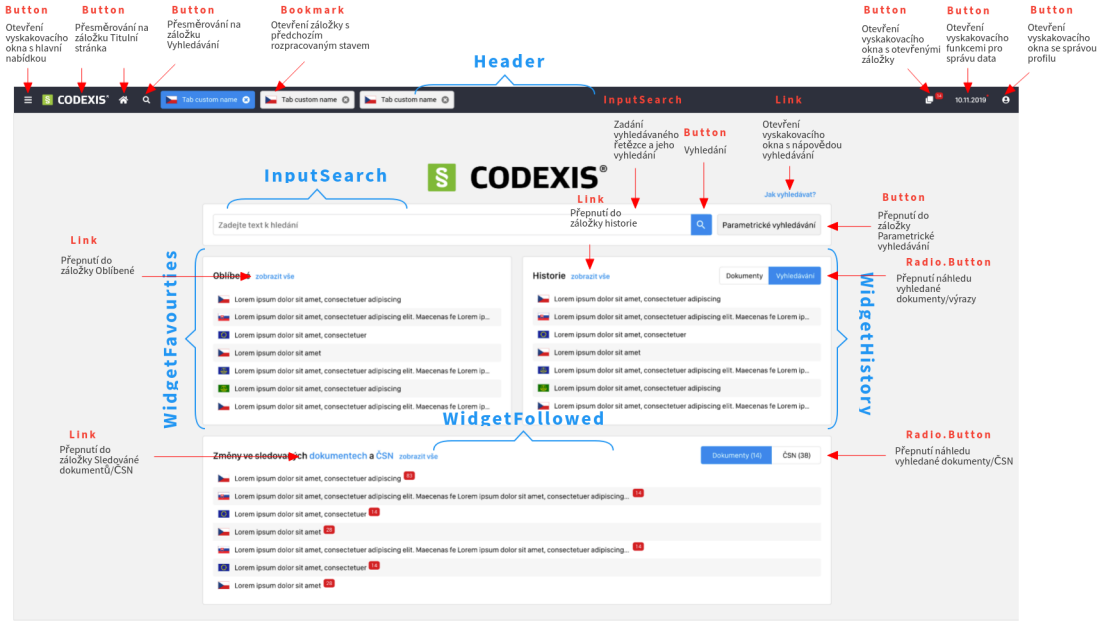
23. *Právní prostor* [online]. ATLAS consulting [cit. 2020-03-04]. Dostupné z: https://www.pravniprostor.cz/.

A Výsledek video chat aplikace

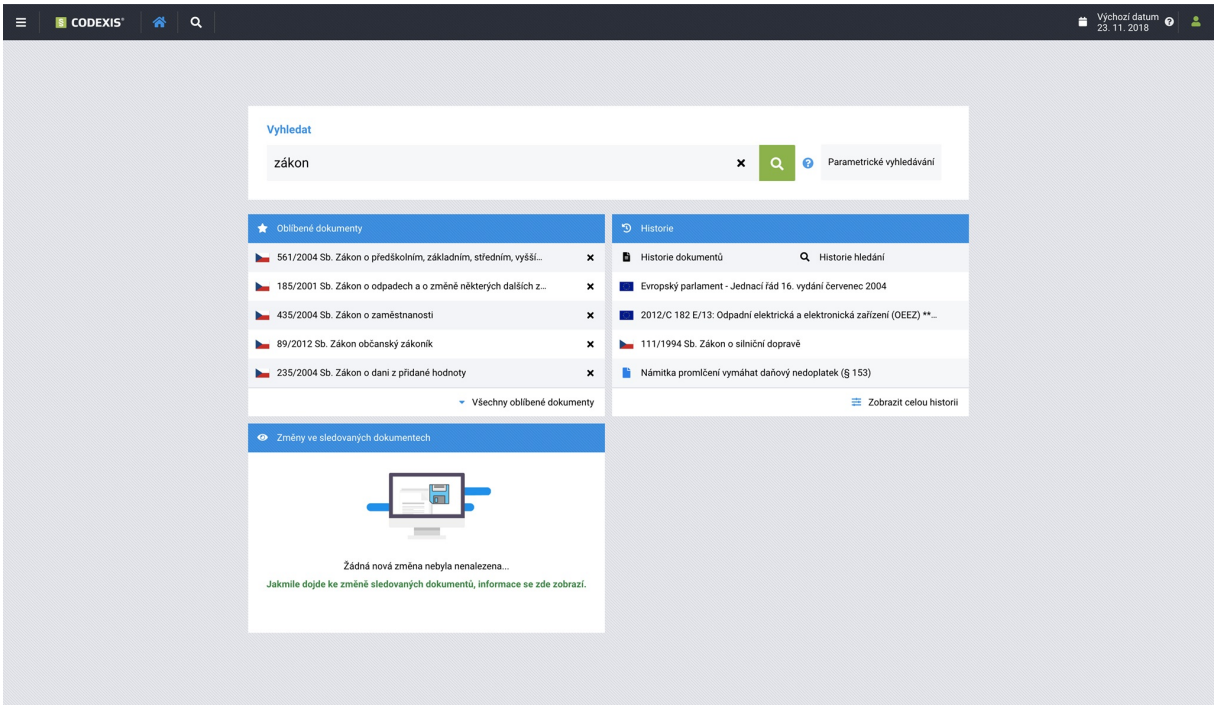


Obrázek 24: Výsledná video chat aplikace

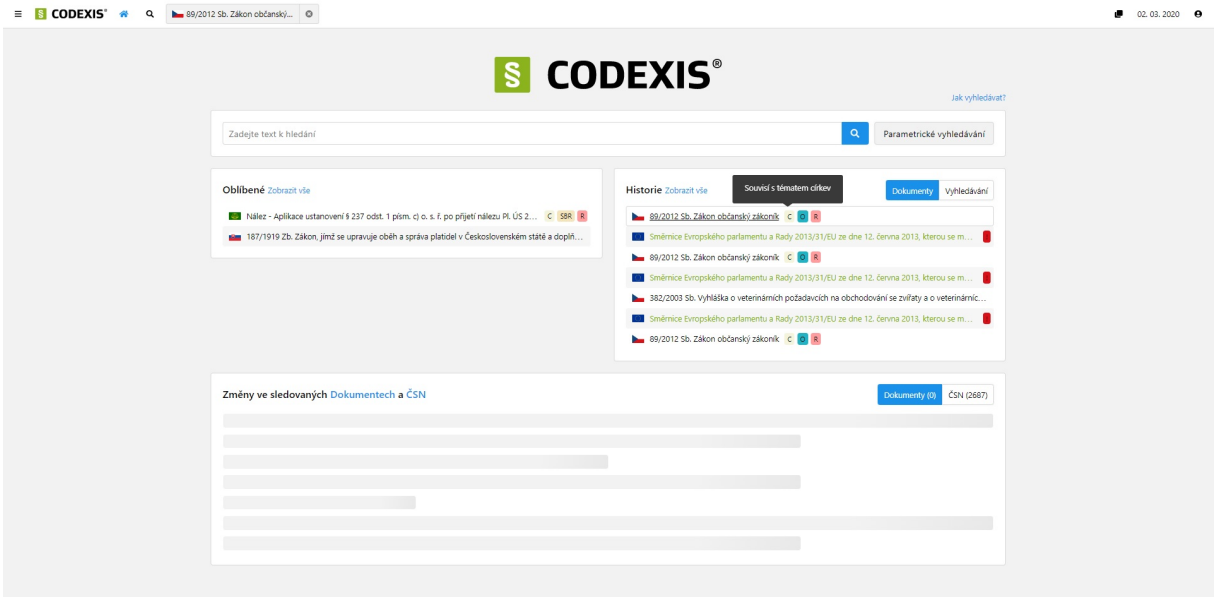
B Grafika pro titulní stránku



Obrázek 25: Grafiký návrh s rozdělením komponent pro záložku - Titulní stránka



Obrázek 26: Stará grafika záložky Titulní stránka



(a) světlé téma

(b) tmavé téma

Obrázek 27: Výsledek záložky Titulní stránka

C Vlastnosti komponent vyvíjených v modulu React-Atlantic

Prop Types

"Skeleton" Component

property	propType	required	default	description
size	Size	-	-	small | medium | large
shape	Shape	-	rectangle	circle | rectangle
className	string	-	-	custom className
width	ReactText	-	-	width
height	number	-	-	height
bgColor	string	-	-	custom background color
animationColors	{ alpha: string; beta: string; gamma: string; }	-	-	custom colors for animation

Obrázek 28: Vlastnosti komponenty Skeleton [22]

Prop Types

"Pagination" Component

property	propType	required	default	description
current	number	-	-	Current page number
defaultCurrent	number	-	1	Default initial page number
isDisabled	boolean	-	-	Disable pagination
pageSize	number	-	20	Number of data per page
pageSizeOptions	number[]	-	[10, 20, 30, 40]	Specify select options
showQuickJumper	boolean	-	-	Determine whether you can jump to pages directly
hideArrowJumper	boolean	-	-	Determine whether you can jump with arrow jumper
showDoubleArrowJumper	boolean	-	-	Determine whether you can jump with double-arrow jumper
showSizeChanger	boolean	-	-	Determine whether pageSize can be changed
showThreeDots	boolean	-	true	Determine whether threedots visible
isSimple	boolean	-	-	Simple mode
total	number	-	0	Total number of data items
className	string	-	-	custom className
onPageChange	(page: number, pageSize: number) => void	-	-	Called when the page number is changed
onSizeChange	(current: number, size: number) => void	-	-	Called when pageSize is changed
tooltipTextRight	string	-	`Next 5 pages`	Tooltip text right
tooltipTextLeft	string	-	`Previous 5 pages`	Tooltip text left
sizeChangerText	string	-	`page`	Select text
quickJumperText	string	-	`Go to`	Jumper text
size	Size	-	medium	small | medium | large

Obrázek 29: Vlastnosti komponenty Pagination [22]

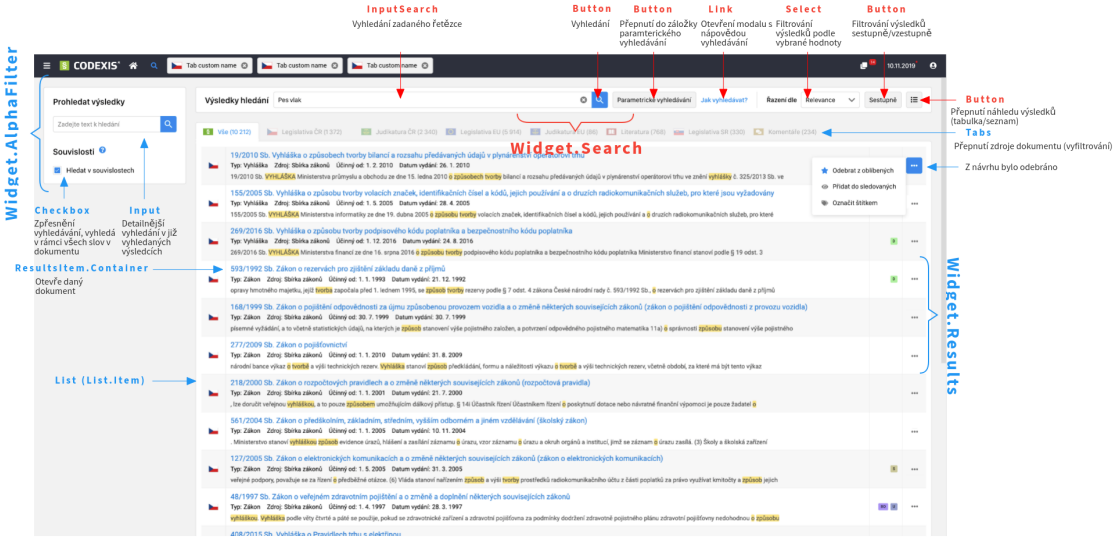
Prop Types

"Carousel" Component

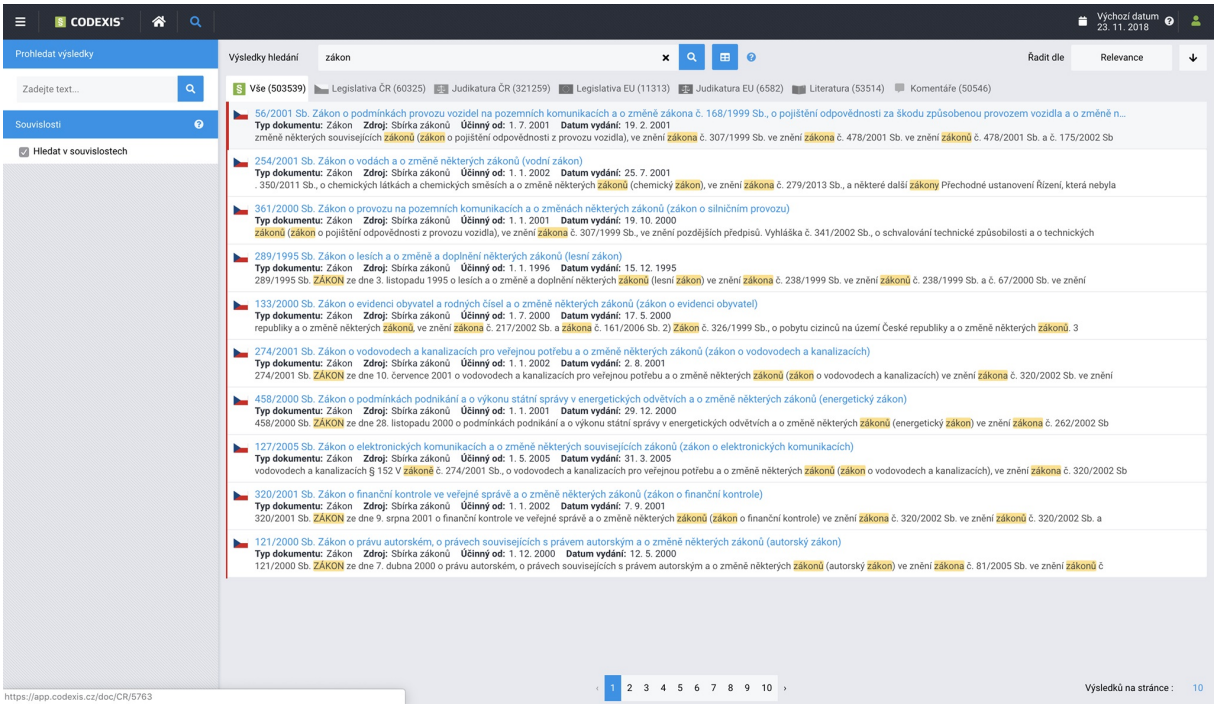
property	propType	required	default	description
activeSlide	number	-	-	
defaultSlide	number	-	0	
auto	number	-	-	
ref	Readonly<HTMLElement>	-	-	
autoHeight	boolean	-	true	
carouselHeight	number	-	-	
onSlideChange	(slide: number) => void	-	-	
springConfig	any	-	-	
onDrag	(setSprings: any, setSlide: (value?: boolean) => void, width: number, index: number, distance?: number, ref?: Readonly<HTMLElement>) => () => Fn | ReactEventHandlers	-	-	

Obrázek 30: Vlastnosti komponenty Carousel [22]

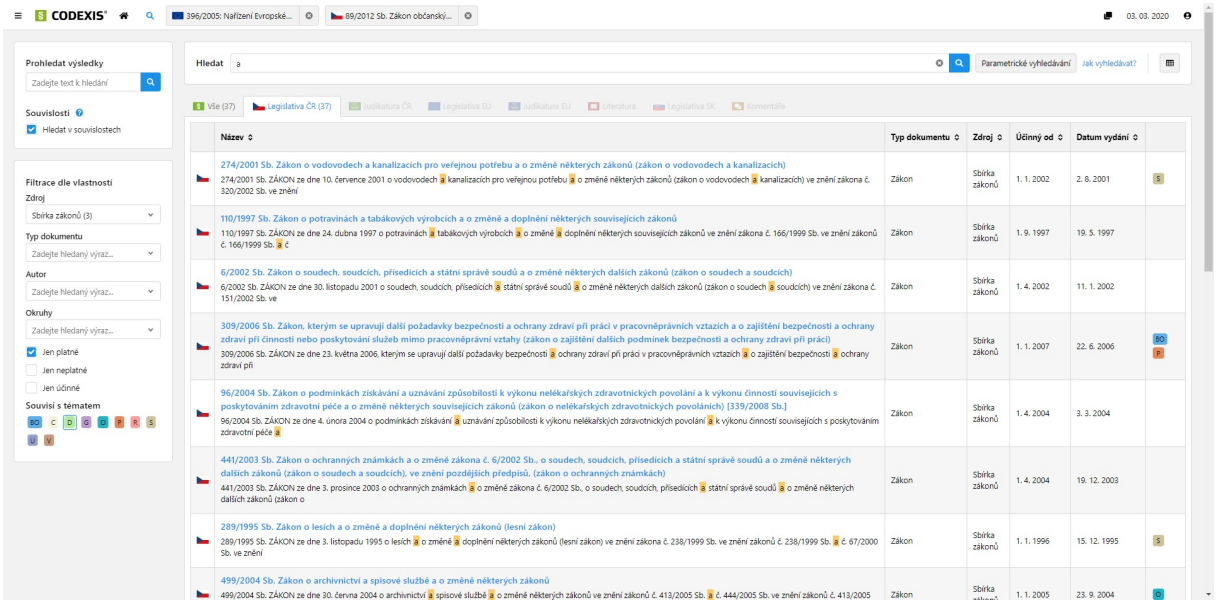
D Grafika pro záložku Vyhledávání



Obrázek 31: Návrh rozdělení komponent pro záložku Vyhledávání (desktopové rozlišení)

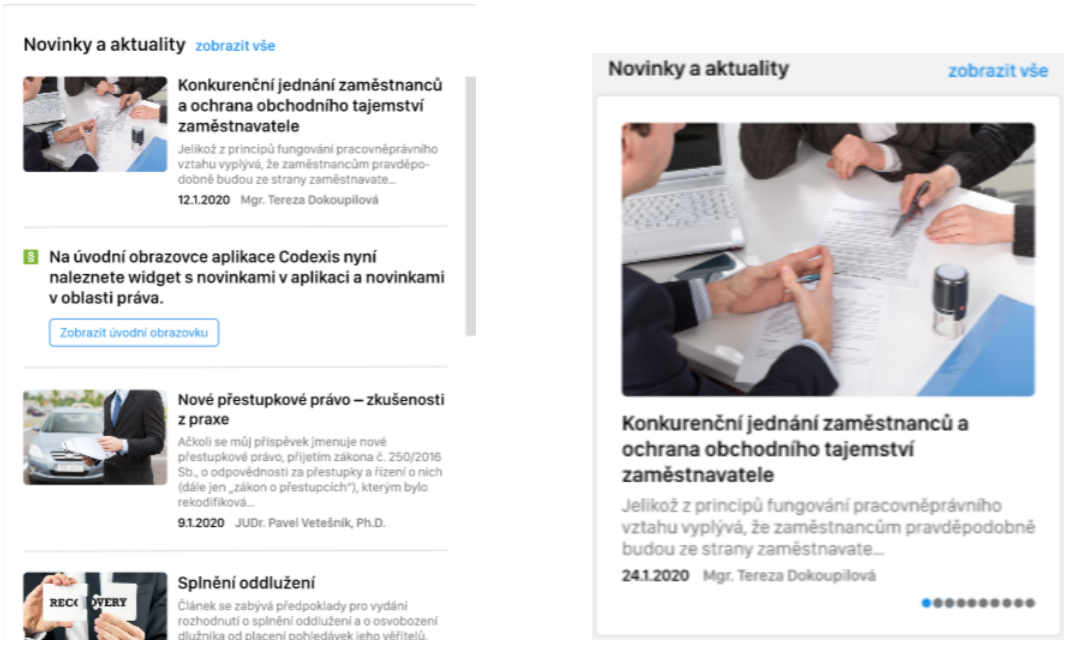


Obrázek 32: Stará grafika záložky Vyhledávání



Obrázek 33: Výsledek přepisu záložky Vyhledávání

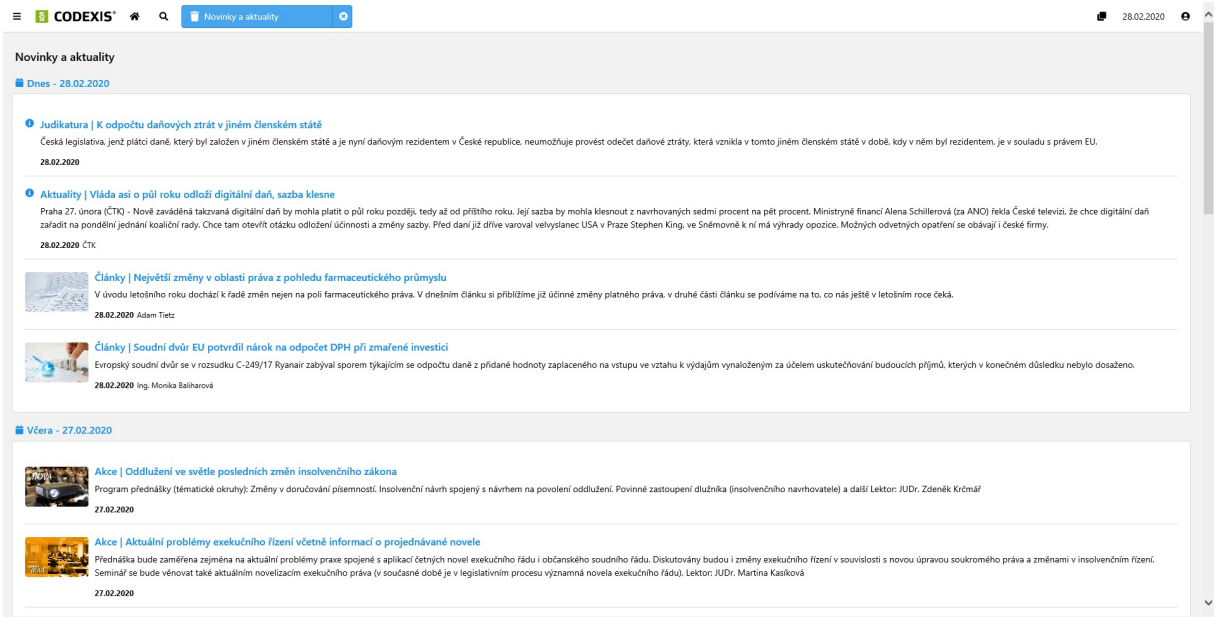
E Grafika pro novinky a aktuality



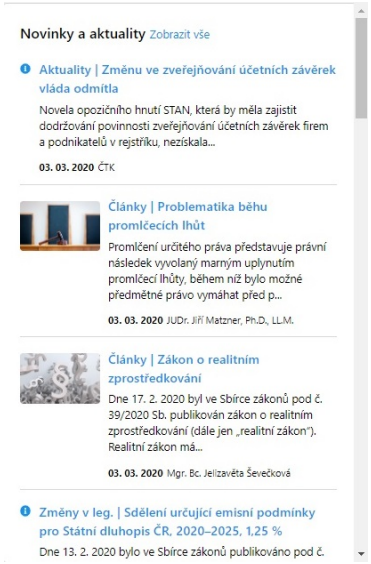
(a) rozlišení pro desktop

(b) rozlišení pro tablet

Obrázek 34: Návrh widgetu s aktualitami na titulní stránce



Obrázek 35: Výsledek záložky - Novinky a aktuality



(a) desktopové rozlišení

(b) mobilní rozlišení

Obrázek 36: Výsledek widgetu s aktualitami na titulní stránce